5

10 # DYNAMIC CONTROL GRAPHS FOR ANALYSIS OF
COORDINATION-CENTRIC SOFTWARE DESIGNS

Related Applications

This application is a continuation of U.S. Provisional Application
15 No. 60/213,496 filed June 23, 2000, incorporated herein by reference.

Technical Field

The present invention relates to static error checking of software systems
designed for execution in a hardware architecture with multiple processing resources.

Background of the Invention

20 A system design and programming methodology is most effective when it is
closely integrated and coheres tightly with its corresponding debugging techniques. In
distributed and embedded system methodologies, the relationship between debugging
approaches and design methodologies has traditionally been one-sided in favor of the
design and programming methodologies. Design and programming methodologies are
25 typically developed without any consideration for the debugging techniques that will
later be applied to software systems designed using that design and programming
methodology. While these typical debugging approaches attempt to exploit features
provided by the design and programming methodologies, the debugging techniques
will normally have little or no impact on what the design and programming features
30 are in the first place. This lack of input from debugging approaches to design and

programming methodologies serves to maintain the role of debugging as an afterthought, even though in a typical system design, debugging consumes a majority of the design time. The need remains for a design and programming methodology that reflects input from, and consideration of, potential debugging approaches in order

5   to enhance the design and reduce the implementation time of software systems.

1.   Packaging of Software Elements

Packaging refers to the set of interfaces a software element presents to other elements in a system. Software packaging has many forms in modern methodologies. Some examples are programming language procedure call interfaces (as with

10   libraries), TCP/IP socket interfaces with scripting languages (as with mail and Web servers), and file formats. Several typical prior art packaging styles are described below, beginning with packaging techniques used in object-oriented programming languages and continuing with a description of more generalized approaches to packaging.

15   A.   Object-Oriented Approaches to Packaging

One common packaging style is based on object-oriented programming languages and provides procedure-based (method-based) packaging for software elements (objects within this framework). These procedure-based packages allow polymorphism (in which several types of objects can have identical interfaces) through

20   subtyping, and code sharing through inheritance (deriving a new class of objects from an already existing class of objects). In a typical object-oriented programming language, an object's interface is defined by the object's methods.

Object-oriented approaches are useful in designing concurrent systems (systems with task level parallelism and multiple processing resources?) because of

25   the availability of active objects (objects with a thread of control). Some common, concurrent object-oriented approaches are shown in actor languages and in concurrent Eiffel.

Early object-oriented approaches featured anonymity of objects through dynamic typechecking. This anonymity of objects meant that a first object did not

need to know anything about a second object in order to send a message to the second object. One unfortunate result of this anonymity of objects was that the second object could unexpectedly respond to the first object that the sent message was not understood, resulting in a lack of predictability, due to this disruption of system

5   executions, for systems designed with this object-oriented approach.

Most modern object-oriented approaches opt to sacrifice the benefits flowing from anonymity of objects in order to facilitate stronger static typing (checking to ensure that objects will properly communicate with one another before actually executing the software system). The main result of stronger static typing is improved

10   system predictability. However, an unfortunate result of sacrificing the anonymity of objects is a tighter coupling between those objects, whereby each object must explicitly classify, and include knowledge about, other objects to which it sends messages. In modern object-oriented approaches the package (interface) has become indistinguishable from the object and the system in which the object is a part.

15   The need remains for a design and programming methodology that combines the benefits of anonymity for the software elements with the benefits derived from strong static typing of system designs.

B.   Other Approaches to Packaging

Other packaging approaches provide higher degrees of separation between

20   software elements and their respective packages than does the packaging in object-oriented systems. For example, the packages in event-based frameworks are interfaces with ports for transmitting and receiving events. These provide loose coupling for interelement communication. However, in an event-based framework, a software designer must explicitly implement interelement state coherence between

25   software elements as communication between those software elements. This means that a programmer must perform the error-prone task of designing, optimizing, implementing, and debugging a specialized communication protocol for each state coherence requirement in a particular software system.

The common object request broker architecture (CORBA) provides an interface description language (IDL) for building packages around software elements written in a variety of languages. These packages are remote procedure call (RPC) based and provide no support for coordinating state between elements. With flexible packaging, an element's package is implemented as a set of co-routines that can be adapted for use with applications through use of adapters with interfaces complementary to the interface for the software element. These adapters can be application-specific—used only when the elements are composed into a system.

The use of co-routines lets a designer specify transactions or sequences of events as part of an interface, rather than just as atomic events. Unfortunately, co-routines must be executed in lock-step, meaning a transition in one routine corresponds to a transition in the other co-routine. If there is an error in one or if an expected event is lost, the interface will fail because its context will be incorrect to recover from the lost event and the co-routines will be out of sync.

The need remains for a design and programming methodology that provides software packaging that supports the implementation of state coherence in distributed concurrent systems without packaging or interface failure when an error or an unexpected event occurs.

2.   Approaches to Coordination

Coordination, within the context of this application, means the predetermined ways through which software components interact. In a broader sense, coordination refers to a methodology for composing concurrent components into a complete system. This use of the term coordination differs slightly from the use of the term in the parallelizing compiler literature, in which coordination refers to a technique for maintaining programwide semantics for a sequential program decomposed into parallel subprograms.

A.   Coordination Languages

Coordination languages are usually a class of tuple-space programming languages, such as Linda. A tuple is a data object containing two or more types of

data that are identified by their tags and parameter lists. In tuple-space languages, coordination occurs through the use of tuple spaces, which are global multisets of tagged tuples stored in shared memory. Tuple-space languages extend existing programming languages by adding six operators: out, in, read, eval, inp, and readp.

5   The out, in, and read operators place, fetch and remove, and fetch without removing tuples from tuple space. Each of these three operators blocks until its operation is complete. The out operator creates tuples containing a tag and several arguments. Procedure calls can be included in the arguments, but since out blocks, the calls must be performed and the results stored in the tuple before the operator can return.

10   The operators eval, inp, and readp are nonblocking versions of out, in, and read, respectively. They increase the expressive power of tuple-space languages. Consider the case of eval, the nonblocking version of out. Instead of evaluating all arguments of the tuple before returning, it spawns a thread to evaluate them, creating, in effect, an active tuple (whereas tuples created by out are passive). As with out,

15   when the computation is finished, the results are stored in a passive tuple and left in tuple space. Unlike out, however, the eval call returns immediately, so that several active tuples can be left outstanding.

Tuple-space coordination can be used in concise implementations of many common interaction protocols. Unfortunately, tuple-space languages do not separate

20   coordination issues from programming issues. Consider the annotated Linda implementation of RPC in Listing 1.

Listing 1: Linda used to emulate RPC:

```
rpcCall(args) {                                          /* C */
    out(" RPCToServer" , " Client" , args...);
    in(" Client, " ReturnFromServer" , &returnValue);
    return returnValue;                                  /* C */
}                                                        /* C */
Server:
...
while(true) {                                            /* C */
    in(" RPCToServer" , &returnAddress, args...);
    returnValue = functionCall(args);                    /* C */
    out(returnAddress, " ReturnFromServer" , returnValue);
}                                                        /* C */
```

6

Although the implementation depicted in Listing 1 is a compact representation of an RPC protocol, the implementation still depends heavily on an accompanying programming language (in this case, C). This dependency prevents designers from creating a new Linda RPC operator for arbitrary applications of RPC. Therefore, every time a designer uses Linda for RPC, they must copy the source code for RPC or make a C-macro. This causes tight coupling, because the client must know the name of the RPC server. If the server name is passed in as a parameter, flexibility increases; however, this requires a binding phase in which the name is obtained and applied outside of the Linda framework.

The need remains for a design and programming methodology that allows implementation of communication protocols without tight coupling between the protocol implementation and the software elements with which the protocol implementation works.

A tuple space can require large quantities of dynamically allocated memory. However, most systems, and especially embedded systems, must operate within predictable and sometimes small memory requirements. Tuple-space systems are usually not suitable for coordination in systems that must operate within small predictable memory requirements because once a tuple has been generated, it remains in tuple space until it is explicitly removed or the software element that created it terminates. Maintaining a global tuple space can be very expensive in terms of overall system performance. Although much work has gone into improving the efficiency of tuple-space languages, system performance remains worse with tuple-space languages than with message-passing techniques.

The need remains for a design and programming methodology that can effectively coordinate between software elements while respecting performance and predictable memory requirements.

B.   Fixed Coordination Models

In tuple-space languages, much of the complexity of coordination remains entangled with the functionality of computational elements. An encapsulating coordination formalism decouples intercomponent interactions from the computational

5   elements.

This type of formalism can be provided by fixed coordination models in which the coordination style is embodied in an entity and separated from computational concerns. Synchronous coordination models coordinate activity through relative schedules. Typically, these approaches require the coordination protocol to be

10   manually constructed in advance. In addition, computational elements must be tailored to the coordination style used for a particular system (which may require intrusive modification of the software elements).

The need remains for a design and programming methodology that allows for coordination between software elements without tailoring the software elements to the

15   specific coordination style used in a particular software system while allowing for interactions between software elements is a way that facilitates debugging complex systems.

Summary of the Invention

Static analysis can be of great benefit in debugging complex systems.

20   Traditional runtime debugging is necessary because certain software errors cannot be detected until after they are compiled into execution errors. Static analysis can reduce the number of such errors and can aid designers by illuminating subtle design interactions. The present invention relates to various types of static analysis that can be applied to coordination-centric software systems, including typechecking,

25   consistency checking, conflict detection through automatically derived abstract views, and model checking. The static analyses presented here comprise a form of preemptive debugging.

Dynamic control graphs (DCGs) lend themselves to a wider variety of dynamic checks and to model checking as well. Model checking enables many more

system checks, such as configuration reachability. To apply model checking to DCGs, it is first necessary to derive a transition relation and represent it as a binary decision diagram (BDD). The size of BDDs is sensitive to variable ordering, but the structure of DCGs enables ordering heuristics that typically result in BDDs of

5   reasonable size.

Static analysis is used not only to find errors or bugs in a system, but to optimize a program as well. This is particularly evident in the case of control dataflow graphs (CDGs). Using these, efficient static schedules can be derived for modal dataflow regions with constant production and consumption rates.

10   Additional aspects and advantages of this invention will be apparent from the following detailed description of preferred embodiments thereof, which proceeds with reference to the accompanying drawings.

Brief Description of the Drawings

Fig. 1 is a component in accordance with the present invention.

15   Fig. 2 is the component of Fig. 1 further having a set of coordination interfaces.

Fig. 3A is a prior art round-robin resource allocation protocol with a centralized controller.

Fig. 3B is a prior art round-robin resource allocation protocol implementing a

20   token passing scheme.

Fig. 4A is a detailed view of a component and a coordination interface connected to the component for use in round-robin resource allocation in accordance with the present invention.

Fig. 4B depicts a round-robin coordinator in accordance with the present

25   invention.

Fig. 5 shows several typical ports for use in a coordination interface in accordance with the present invention.

Fig. 6A is a unidirectional data transfer coordinator in accordance with the present invention.

Fig. 6B is a bidirectional data transfer coordinator in accordance with the present invention.

5   Fig. 6C is a state unification coordinator in accordance with the present invention.

Fig. 6D is a control state mutex coordinator in accordance with the present invention.

Fig. 7 is a system for implementing subsumption resource allocation having

10   components, a shared resource, and a subsumption coordinator.

Fig. 8 is a barrier synchronization coordinator in accordance with the present invention.

Fig. 9 is a rendezvous coordinator in accordance with the present invention.

Fig. 10 depicts a dedicated RPC system having a client, a server, and a

15   dedicated RPC coordinator coordinating the activities of the client and the server.

Fig. 11 is a compound coordinator with both preemption and round-robin coordination for controlling the access of a set of components to a shared resource.

Fig. 12A is software system with two data transfer coordinators, each having constant message consumption and generation rules and each connected to a separate

20   data-generating component and connected to the same data-receiving component.

Fig. 12B is the software system of Fig. 12A in which the two data transfer coordinators have been replaced with a merged data transfer coordinator.

Fig. 13 is a system implementing a first come, first served resource allocation protocol in accordance with the present invention.

25   Fig. 14 is a system implementing a multiclient RPC coordination protocol formed by combining the first come, first served protocol of Fig. 13 with the dedicated RPC coordinator of Fig. 10.

Fig. 15 depicts a large system in which the coordination-centric design methodology can be employed having a wireless device interacting with a cellular network.

Fig. 16 shows a top-level view of the behavior and components for a system
5   for a cell phone.

Fig. 17A is a detailed view of a GUI component of the cell phone of Fig. 16.

Fig. 17B is a detailed view of a call log component of the cell phone of Fig. 16.

Fig. 18A is a detailed view of a voice subsystem component of the cell phone
10   of Fig. 16.

Fig. 18B is a detailed view of a connection component of the cell phone of Fig. 16.

Fig. 19 depicts the coordination layers between a wireless device and a base station, and between the base station and a switching center, of Fig. 15.

15   Fig. 20 depicts a cell phone call management component, a master switching center call management component, and a call management coordinator connecting the respective call management components.

Fig. 21A is a detailed view of a transport component of the connection component of Fig. 18B.

20   Fig. 21B is a CDMA data modulator of the transport component of Fig. 18B.

Fig. 22 is a detailed view of a typical TDMA and a typical CDMA signal for the cell phone of Fig. 16.

Fig. 23A is a LCD touch screen component for a Web browser GUI for a wireless device.

25   Fig. 23B is a Web page formatter component for the Web browser GUI for the wireless device.

Fig. 24A is a completed GUI system for a handheld Web browser.

Fig. 24B shows the GUI system for the handheld Web browser combined with the connection subsystem of Fig. 18B in order to access the cellular network of Fig. 15.

Fig. 25 is a typical space/time diagram with space represented on a vertical axis and time represented on a horizontal axis.

Fig. 26 is a space/time diagram depicting a set of system events and two different observations of those system events.

Fig. 27 is a space/time diagram depicting a set of system events and an ideal observation of the events taken by a real-time observer.

Fig. 28 is a space/time diagram depicting two different yet valid observations of a system execution.

Fig. 29 is a space/time diagram depicting a system execution and an observation of that execution take by a discrete lamport observer.

Fig. 30 is a space/time diagram depicting a set of events that each include a lamport time stamp.

Fig. 31 is a space/time diagram illustrating the insufficiency of scalar timestamps to characterize causality between events.

Fig. 32 is a space/time diagram depicting a set of system events that each a vector time stamp.

Fig. 33 depicts a display from a Partial Order Event Tracer (POET).

Fig. 34 is a space/time diagram depicting two compound events that are neither causal nor concurrent.

Fig. 35 is a POET display of two convex event clusters.

Fig. 36 is a basis for distributed event environments (BEE) abstraction facility for a single client.

Fig. 37 is a hierarchical tree construction of process clusters.

Fig. 38A depicts a qualitative measure of cohesion and coupling between a set of process clusters that have heavy communication or are instantiated from the same source code.

Fig. 38B depicts a qualitative measure of cohesion and coupling between a set of process clusters that do not have heavy communication or are not instances of the same source code.

Fig. 38C depicts a qualitative measure of cohesion and coupling between an alternative set of process clusters that have heavy communication or are instantiated from the same source code.

Fig. 39 depicts a consistent and an inconsistent cut of a system execution on a space/time diagram.

Fig. 40A is a space/time diagram depicting a system execution.

Fig. 40B is a lattice representing all possible consistent cuts of the space/time diagram of Fig. 40A.

Fig. 40C is a graphical representation of the possible consistent cuts of Fig. 40B.

Fig. 41A is a space/time diagram depicting a system execution.

Fig. 41B is the space/time diagram of Fig. 41A after performing a global-step.

Fig. 41C is the space/time diagram of Fig. 41A after performing a step-over.

Fig. 41D is the space/time diagram of Fig. 41A after performing a step-in.

Fig. 42 is a space/time diagram depicting a system that is subject to a domino effect whenever the system is rolled back in time to a checkpoint.

Fig. 43 depicts a simple static control graph in accordance with the present invention.

Fig. 44A is an edge that asserts the value true at its head and is responsive to the value true at its tail.

Fig. 44B is an edge that asserts the value true at its head and is responsive to the value false at its tail.

Fig. 44C is an edge that asserts the value false at its head and is responsive to the value true at its tail.

5   Fig. 44D is an edge that asserts the value false at its head and is responsive to the value false at its tail.

Fig. 45 is an illustration of the semantic differences between Boolean networks and static control graphs.

Fig. 46A shows a basic control graph with reduced characteristic functions.

10   Fig. 46B shows a basic control graph with reduced characteristic functions.

Fig. 47A shows the impact of edge semantics on SCG semantics when an enforcing edge asserts a value of false at its head.

Fig. 47B shows the impact of edge semantics on SCG semantics when an enforcing edge is responsive to a value of false at its tail.

15   Fig. 47C shows the impact of edge semantics on SCG semantics when a sensing edge asserts a value of false at its head.

Fig. 47D shows the impact of edge semantics on SCG semantics when a sensing edge is responsive to a value of false at its tail.

Fig. 48A depicts a coordinator for a rendezvous-style coordination.

20   Fig. 48B is the static control graph that represents a coordinator for a rendezvous-style coordination..

Fig. 49 is a static control graph with no stable and non-conflicting states.

Fig. 50A is the static control graph that represents the reduction from 3-SAT to Any Stable State Property.

25   Fig. 50B is a logic representation for the static control graph that represents the reduction from 3-SAT to Any Stable State Property.

Fig. 50C is an equivalent logic representation for the static control graph that represents the reduction from 3-SAT to Any Stable State Property.

Fig. 51A is a static control graph showing a first order conflict.

Fig. 51B is a static control graph showing a second order conflict.

Fig. 52A is a static control graph showing a disjunctive node (d_i) before "flattening."

Fig. 52B is the static control graph showing the disjunctive node (d_i) after "flattening."

Fig. 53 is a static control graph that shows a potential conflict, highlighted by "flattening."

Fig. 54 is the static control graph for Figure 49, after "flattening."

Fig. 55A shows a component's inner static control graph.

Fig. 55B shows the component's static control graph after the internal nodes are summarized as a single, independent node.

Fig. 56 shows an ideal structure for applying hierarchical reduction to highlight instability in a system.

Fig. 57 depicts a dynamic control graph (DCG).

Fig. 58 depicts a DCG with an action node.

Fig. 59 shows a DCG with an action node for an action that is transparent with respect to control interactions.

Fig. 60 shows a DCG with an action node for an action that is opaque with respect to control interactions.

Fig. 61A shows a DCG for a rendezvous coordinator

Fig. 61B shows a DCG for a rendezvous coordinator with two participant preemption.

Fig. 62A shows a communication channel between partitions of a static control graph (SCG) that can cause an action-only barrier.

Fig. 62B shows a DCG corresponding to the SCG of Fig. 62A after partitioning across the access-only barrier.

5   Fig. 63, depicts constraint edges that cross the action only barrier, from Fig. 62A, and their corresponding templates.

Fig. 64 depicts a current DCG along with a next DCG, which is the result of temporally unrolling DCG.

Fig. 65A depicts a simple DCG.

10   Fig. 65B depicts an unrolled DCG for simple DCG.

Fig. 66A shows a truth table for a boolean "and" function.

Fig. 66B shows a truth tree that corresponds to the truth table in Fig. 66A.

Fig. 67A shows a reduced binary decision diagram (BDD) for the truth tree in Fig. 66B.

15   Fig. 67B shows an alternate reduced BDD for the truth tree in Fig. 66B.

Fig. 67C shows a third alternate reduced BDD for the truth tree in Fig. 66B.

Fig. 68 shows the results of using the apply algorithm to grow a BDD, which represents the characteristic function of the unrolled DCG from Fig. 65B.

Fig. 69A shows an unrolled DCG for the rendezvous DCG of Fig. 61A.

20   Fig. 69B shows that a critical factor in ordering for the characteristic function of the unrolled DCG of Fig. 69A is the relative ordering of $wait_c$, $wait_b$, and $wait_a$.

Figs. 70A, B, C, and D show that interleaving asynchronous models is not a sufficient static error-checking tool for coordination-centric system designs.

Fig. 71 shows a control/dataflow graph (CDG).

25   Fig. 72 shows a CDG representation of an RPC system.

Fig. 73A shows the second step in partitioning a CDG across an action-only barrier.

Fig. 73B shows the graph of DCG from Fig. 62B, with the action-only barrier transformed to a message-only barrier.

5   Fig. 74A shows a CDG with a set of message rate guarantees.

Fig. 74B shows a dataflow graph based on the CDG of Fig. 74A.

Fig. 75 shows a dataflow graph.

Detailed Description of Preferred Embodiment

Coordination-Centric Software Design

10   Fig. 1 is an example of a component 100, which is the basic software element within the coordination-centric design framework, in accordance with the present invention. With reference to Fig. 1, component 100 contains a set of modes 102. Each mode 102 corresponds to a specific behavior associated with component 100. Each mode 102 can either be active or inactive, respectively enabling or disabling the

15   behavior corresponding to that mode 102. Modes 102 can make the conditional aspects of the behavior of component 100 explicit. The behavior of component 100 is encapsulated in a set of actions 104, which are discrete, event-triggered behavioral elements within the coordination-centric design methodology. Component 100 can be copied and the copies of component 100 can be modified, providing the code-sharing

20   benefits of inheritance.

Actions 104 are enabled and disabled by modes 102, and hence can be thought of as effectively being properties of modes 102. An event (not shown) is an instantaneous condition, such as a timer tick, a data departure or arrival, or a mode change. Actions 104 can activate and deactivate modes 102, thereby selecting the

25   future behavior of component 100. This is similar to actor languages, in which methods are allowed to replace an object's behavior.

In coordination-centric design, however, all possible behaviors must be identified and encapsulated before runtime. For example, a designer building a user interface component for a cell phone might define one mode for looking up numbers

in an address book (in which the user interface behavior is to display complete address book entries in formatted text) and another mode for displaying the status of the phone (in which the user interface behavior is to graphically display the signal power and the battery levels of the phone). The designer must define both the modes

5   and the actions for the given behaviors well before the component can be executed.

Fig. 2 is component 100 further including a first coordination interface 200, a second coordination interface 202, and a third coordination interface 204. Coordination-centric design's components 100 provide the code-sharing capability of object-oriented inheritance through copying. Another aspect of object-oriented

10   inheritance is polymorphism through shared interfaces. In object-oriented languages, an object's interface is defined by its methods. Although coordination-centric design's actions 104 are similar to methods in object-oriented languages, they do not define the interface for component 100. Components interact through explicit and separate coordination interfaces, in this figure coordination interfaces 200, 202, and

15   204. The shape of coordination interfaces 200, 202, and 204 determines the ways in which component 100 may be connected within a software system. The way coordination interfaces 200, 202, and 204 are connected to modes 102 and actions 104 within component 100 determines how the behavior of component 100 can be managed within a system. Systemwide behavior is managed through coordinators

20   (see Fig. 4B and subsequent).

For our approach to be effective, several factors in the design of software elements must coincide: packaging, internal organization, and how elements coordinate their behavior. Although these are often treated as independent issues, conflicts among them can exacerbate debugging. We handle them in a unified

25   framework that separates the internal activity from the external relationship of component 100. This lets designers build more modular components and encourages them to specify distributable versions of coordination protocols. Components can be reused in a variety of contexts, both distributed, and single processor 1.

1.   Introduction to Coordination

Within this application, coordination refers to the predetermined ways by which components interact. Consider a common coordination activity: resource allocation. One simple protocol for this is round-robin: participants are lined up, and the resource is given to each participant in turn. After the last participant is served, the resource is given back to the first. There is a resource-scheduling period during which each participant gets the resource exactly once, whether or not it is needed.

Fig. 3A is prior art round-robin resource allocation protocol with a centralized controller 300, which keeps track of and distributes the shared resource (not shown) to each of software elements 302, 304, 306, 308, and 310 in turn. With reference to Fig. 3A, controller 300 alone determines which software element 302, 304, 306, 308, or 310 is currently allowed to use the resource and which has it next. This implementation of a round-robin protocol permits software elements 302, 304, 306, 308, and 310 to be modular, because only controller 300 keeps track of the software elements. Unfortunately, when this implementation is implemented on a distributed architecture (not shown), controller 300 must typically be placed on a single processing element (not shown). As a result, all coordination requests must go through that processing element, which can cause a communication performance bottleneck. For example, consider the situation in which software elements 304 and 306 are implemented on a first processing element (not shown) and controller 300 is implemented on a second processing element. Software element 304 releases the shared resource and must send a message indicating this to controller 300. Controller 300 must then send a message to software element 306 to inform software element 306 that it now has the right to the shared resource. If the communication channel between the first processing resource and the second processing resource is in use or the second processing element is busy, then the shared resource must remain idle, even though both the current resource holder and the next resource holder (software elements 304 and 306 respectively) are implemented on the first processing element (not shown). The shared resource must typically remain idle until communication can

take place and controller 300 can respond. This is an inefficient way to control access to a shared resource.

      Fig. 3B is a prior art round-robin resource allocation protocol implementing a token passing scheme. With reference to Fig. 3B, this system consists of a shared

5   resource 311 and a set of software elements 312, 314, 316, 318, 320, and 322. In this system a logical token 324 symbolizes the right to access resource 311, *i.e.,* when a software element holds token 324, it has the right to access resource 311. When one of software elements 312, 314, 316, 318, 320, or 322 finishes with resource 311, it passes token 324, and with token 324 the access right, to a successor. This

10   implementation can be distributed without a centralized controller, but as shown in Figure v3B, this is less modular, because it requires each software element in the set to keep track of a successor.

      Not only must software elements 312, 314, 316, 318, 320, and 322 keep track of successors, but each must implement a potentially complicated and error-prone

15   protocol for transferring token 324 to its successor. Bugs can cause token 324 to be lost or introduce multiple tokens 324. Since there is no formal connection between the physical system and complete topology maps (diagrams that show how each software element is connected to others within the system), some software elements might erroneously be serviced more than once per cycle, while others are completely

20   neglected. However, these bugs can be extremely difficult to track after the system is completed. The protocol is entangled with the functionality of each software element, and it is difficult to separate the two for debugging purposes. Furthermore, if a few of the software elements are located on the same machine, performance of the implementation can be poor. The entangling of computation and coordination

25   requires intrusive modification to optimize the system.

2.   Coordination-Centric Design's Approach to Coordination

      The coordination-centric design methodology provides an encapsulating formalism for coordination. Components such as component 100 interact using coordination interfaces, such as first, second, and third coordination interfaces 200,

30   202, and 204, respectively. Coordination interfaces preserve component modularity

while exposing any parts of a component that participate in coordination. This technique of connecting components provides polymorphism in a similar fashion to subtyping in object-oriented languages.

5   Fig. 4A is a detailed view of a component 400 and a resource access coordination interface 402 connected to component 400 for use in a round-robin coordination protocol in accordance with the present invention. With reference to Fig. 4A, resource access coordination interface 402 facilitates implementation of a round-robin protocol that is similar to the token-passing round-robin protocol described above. Resource access coordination interface 402 has a single bit of

10   control state, called access, which is shown as an arbitrated control port 404 that indicates whether or not component 400 is holding a virtual token (not shown). Component 400 can only use a send message port 406 on access coordination interface 402 when arbitrated control port 404 is true. Access coordination interface 402 further has a receive message port 408.

15   Fig. 4B show a round-robin coordinator 410 in accordance with the present invention. With reference to Fig. 4B, round-robin coordinator 410 has a set of coordinator coordination interfaces 412 for connecting to a set of components 400. Each component 400 includes a resource access coordination interface 402. Each coordinator coordination interface 412 has a coordinator arbitrated control port 414,

20   an incoming send message port 416 and an outgoing receive message port 418. Coordinator coordination interface 412 in complimentary to resource access coordination interface 402, and vice versa, because the ports on the two interfaces are compatible and can function to transfer information between the two interfaces.

The round-robin protocol requires round-robin coordinator 410 to manage the

25   coordination topology. Round-robin coordinator 410 is an instance of more general abstractions called coordination classes, in which coordination classes define specific coordination protocols and a coordinator is a specific implementation of the coordination class. Round-robin coordinator 410 contains all information about how components 400 are supposed to coordinate. Although round-robin coordinator 410

30   can have a distributed implementation, no component 400 is required to keep

references to any other component 400 (unlike the distributed round-robin implementation shown in Fig. 3B). All required references are maintained by round-robin coordinator 410 itself, and components 400 do not even need to know that they are coordinating through round-robin. Resource access coordination interface 402 can

5   be used with any coordinator that provides the appropriate complementary interface. A coordinator's design is independent of whether it is implemented on a distributed platform or on a monolithic single processor platform.

3.   Coordination Interfaces

Coordination interfaces are used to connect components to coordinators. They

10   are also the principle key to a variety of useful runtime debugging techniques. Coordination interfaces support component modularity by exposing all parts of the component that participate in the coordination protocol. Ports are elements of coordination interfaces, as are guarantees and requirements, each of which will be described in turn.

15   A.   Ports

A port is a primitive connection point for interconnecting components. Each port is a five-tuple (T; A; Q; D; R) in which:

- T represents the data type of the port. T can be one of int, boolean, char, byte, float, double, or cluster, in which cluster represents a cluster of

20   data types (e.g., an int followed by a float followed by two bytes).

- A is a boolean value that is true if the port is arbitrated and false otherwise.

- Q is an integer greater than zero that represents logical queue depth for a port.

25   - D is one of in, out, inout, or custom and represents the direction data flows with respect to the port.

- R is one of discard-on-read, discard-on-transfer, or hold and represents the policy for data removal on the port. Discard-on-read indicates that data is removed immediately after it is read (and any data in the logical

30   queue are shifted), discard-on-transfer indicates that data is removed from

a port immediately after being transferred to another port, and hold indicates that data should be held until it is overwritten by another value. Hold is subject to arbitration.

Custom directionality allows designers to specify ports that accept or generate only certain specific values. For example, a designer may want a port that allows other components to activate, but not deactivate, a mode. While many combinations of port attributes are possible, we normally encounter only a few. The three most common are message ports (output or input), state ports (output, input, or both; sometimes arbitrated), and control ports (a type of state port). Fig. 5 illustrates the visual syntax used for several common ports throughout this application. With reference to Fig. 5, this figure depicts an exported state port 502, an imported state port 504, an arbitrated state port 506, an output data port 508, and an input data port 510.

1.   Message Ports

Message ports (output and input) data ports 508 and 510 respectively) are either send (T; false; 1; out; discard-on-transfer) or receive (T; false; Q; in; discard-on-read). Their function is to transfer data between components. Data passed to a send port is transferred immediately to the corresponding receive port, thus it cannot be retrieved from the send port later. Receive data ports can have queues of various depths. Data arrivals on these ports are frequently used to trigger and pass data parameters into actions. Values remain on receive ports until they are read.

2.   State Ports

State ports take one of three forms:

1. (T; false; 1; out; hold)

2. (T; false; 1; in; hold)

3. (T; true; 1; inout; hold)

State ports, such as exported state port 502, imported state port 504, and arbitrated state port 506, hold persistent values, and the value assigned to a state port may be arbitrated. This means that, unlike message ports, values remain on the state ports until changed. When multiple software elements simultaneously attempt to alter

the value of arbitrated state port 506, the final value is determined based on arbitration rules provided by the designer through an arbitration coordinator (not shown).

State ports transfer variable values between scopes, as explained below. In coordination-centric design, all variables referenced by a component are local to that component, and these variables must be explicitly declared in the component's scope. Variables can, however, be bound to state ports that are connected to other components. In this way a variable value can be transferred between components and the variable value achieves the system-level effect of a multivariable.

3.   Control Ports

Control ports are similar to state ports, but a control port is limited to having the boolean data type. Control ports are typically bound to modes. Actions interact with a control port indirectly, by setting and responding to the values of a mode that is bound to the control port.

For example, arbitrated control port 404 shown in Fig. 4A is a control port that can be bound to a mode (not shown) containing all actions that send data on a shared channel. When arbitrated control port 404 is false, the mode is inactive, disabling all actions that send data on the channel.

B.   Guarantees

Guarantees are formal declarations of invariant properties of a coordination interface. There can be several types of guarantees, such as timing guarantees between events, guarantees between control state (*e.g.,* state A and state B are guaranteed to be mutually exclusive), etc. Although a coordination interface's guarantees reflect properties of the component to which the coordination interface is connected, the guarantees are not physically bound to any internal portions of the component. Guarantees can often be certified through static analysis of the software system. Guarantees are meant to cache various properties that are inherent in a component or a coordinator in order to simplify static analysis of the software system.

A guarantee is a promise provided by a coordination interface. The guarantee takes the form of a predicate promised to be invariant. In principle, guarantees can

include any type of predicate (*e.g.,* x > 3, in which x is an integer valued state port, or $t_{ea} - t_{eb} < 2ms$). Throughout the remainder of this application, guarantees will be only event-ordering guarantees (guarantees that specify acceptable orders of events) or control-relationship guarantees (guarantees pertaining to acceptable relative

5   component behaviors).

C.   Requirements

A requirement is a formal declaration of the properties necessary for correct software system functionality. An example of a requirement is a required response time for a coordination interface—the number of messages that must have arrived at

10   the coordination interface before the coordination interface can transmit, or fire, the messages. When two coordination interfaces are bound together, the requirements of the first coordination interface must be conservatively matched by the guarantees of the second coordination interface (*e.g.,* x < 7 as a guarantee conservatively matches x < 8 as a requirement). As with guarantees, requirements are not physically bound

15   to anything within the component itself. Guarantees can often be verified to be sufficient for the correct operation of the software system in which the component is used. In sum, a requirement is a predicate on a first coordination interface that must be conservatively matched with a guarantee on a complementary second coordination interface.

20   ### D.   Conclusion Regarding Coordination Interfaces

A coordination interface is a four-tuple (P; G; R; I) in which:

- P is a set of named ports.

- G is a set of named guarantees provided by the interface.

- R is a set of named requirements that must be matched by guarantees of

25   connected interfaces.

- I is a set of named coordination interfaces.

As this definition shows, coordination interfaces are recursive. Coordinator coordination interface 412, shown in Fig. 4B, used for round-robin coordination is called *AccessInterface* and is defined in Table 1.

30

Constituent	Value
ports	$P = \{$ access:StatePort, s:outMessagePort, r:inMessagePort $\}$
guarantees	$G = \{ \neg$ access $\Rightarrow \neg$ s.gen $\}$
requirements	$R = \varnothing$
interfaces	$I = \varnothing$

Related to coordination interfaces is a recursive coordination interface descriptor, which is a five-tuple $(P_a; G_a; R_a; I_d; N_d)$ in which:

- P_a is a set of abstract ports, which are ports that may be incomplete in their attributes (*i.e.*, they do not yet have a datatype).

- G_a is a set of abstract guarantees, which are guarantees between abstract ports.

- R_a is a set of abstract requirements, which are requirements between abstract ports.

- I_d is a set of coordination interface descriptors.

- N_d is an element of Q x Q, where Q $= \{\infty\} \cup$ Z+ and Z+ denotes the set of positive integers. N_d indicates the number or range of numbers of permissible interfaces.

Allowing coordination interfaces to contain other coordination interfaces is a powerful feature. It lets designers use common coordination interfaces as complex ports within other coordination interfaces. For example, the basic message ports described above are nonblocking, but we can build a blocking coordination interface (not shown) that serves as a blocking port by combining a wait state port with a message port.

4.   Coordinators

A coordinator provides the concrete representations of intercomponent aspects of a coordination protocol. Coordinators allow a variety of static analysis debugging methodologies for software systems created with the coordination-centric design

methodology. A coordinator contains a set of coordination interfaces and defines the relationships the coordination interfaces. The coordination interfaces complement the component coordination interfaces provided by components operating within the protocol. Through matched interface pairs, coordinators effectively describe

5   connections between message ports, correlations between control states, and transactions between components.

For example, round-robin coordinator 410, shown in Fig. 4B, must ensure that only one component 400 has its component control port 404's value, or its access bit, set to true. Round-robin coordinator 410 must further ensure that the correct

10   component 400 has its component control port 404 set to true for the chosen sequence. This section presents formal definitions of the parts that comprise coordinators: modes, actions, bindings, action triples, and constraints. These definitions culminate in a formal definition of coordinators.

A.   Modes

15   A mode is a boolean value that can be used as a guard on an action. In a coordinator, the mode is most often bound to a control port in a coordination interface for the coordinator. For example, in round-robin coordinator 410, the modes of concern are bound to a coordinator control port 414 of each coordinator coordination interface 412.

20   B.   Actions

An action is a primitive behavioral element that can:

• Respond to events.

• Generate events.

• Change modes.

25   Actions can range in complexity from simple operations up to complicated pieces of source code. An action in a coordinator is called a transparent action because the effects of the action can be precomputed and the internals of the action are completely exposed to the coordination-centric design tools.

C.   Bindings

Bindings connect input ports to output ports, control ports to modes, state ports to variables, and message ports to events. Bindings are transparent and passive. Bindings are simply conduits for event notification and data transfer. When used for event notification, bindings are called triggers.

D.   Action Triples

To be executed, an action must be enabled by a mode and triggered by an event. The combination of a mode, trigger, and action is referred to as an action triple, which is a triple (m; t; a) in which:

- m is a mode.

- t is a trigger.

- a is an action.

The trigger is a reference to an event type, but it can be used to pass data into the action. Action triples are written: mode : trigger : action

A coordinator's actions are usually either pure control, in which both the trigger and action performed affect only control state, or pure data, in which both the trigger and action performed occur in the data domain. In the case of round-robin coordinator 410, the following set of actions is responsible for maintaining the appropriate state:

$$access_i \ : - access_i : \ +access_{(i+1) \ mod \ n}$$

The symbol "+" signifies a mode's activation edge *(i.e.,* the event associated with the mode becoming true), and the symbol "-" signifies its deactivation edge. When any coordinator coordination interface 412 deactivates its arbitrated control port 404's, access bit, the access bit of the next coordinator coordination interface 412 is automatically activated.

E.   Constraints

In this dissertation, constraints are boolean relationships between control ports. They take the form:

$$\text{Condition} \quad \Rightarrow \quad \text{Effect}$$

This essentially means that the Condition (on the left side of the arrow) being true
implies that Effect (on the right side of the arrow) is also true. In other words, if

5   Condition is true, then Effect should also be true.

A constraint differs from a guarantee in that the guarantee is limited to
communicating in-variant relationships between components without providing a way
to enforce the in-variant relationship. The constraint, on the other hand, is a set of
instructions to the runtime system dealing with how to enforce certain relationships

10   between components. When a constraint is violated, two corrective actions are
available to the system: (1) modify the values on the left-hand side to make the
left-hand expression evaluate as false (an effect sometimes termed backpressure) or
(2) alter the right-hand side to make it true. We refer to these techniques as LHM
(left-hand modify) and RHM (right-hand modify). For example, given the constraint

15     $x \Rightarrow \neg y$ and the value $x \wedge y$, with RHM semantics the runtime system must respond
by disabling y or setting y to false. Thus the value of $\neg y$ is set to true.

The decision of whether to use LHM, to use RHM, or even to suspend
enforcement of a constraint in certain situations can dramatically affect the efficiency
and predictability of the software system. Coordination-centric design does not

20   attempt to solve simultaneous constraints at runtime. Rather, runtime algorithms use
local ordered constraint solutions. This, however, can result in some constraints
being violated and is discussed further below.

Round-robin coordinator 410 has a set of safety constraints to ensure that there
is never more than one token in the system:

25

$$\text{access}_i \quad \Rightarrow \quad \forall_{j \neq i} \neg \text{access}_j$$

The above equation translates roughly as access_i implies not access_j for the set of all
access_j where j is not equal to i. Even this simple constraint system can cause

30   problems with local resolution semantics (as are LHM and RHM). If the runtime

system attempted to fix all constraints simultaneously, all access modes would be shut down. If they were fixed one at a time, however, any duplicate tokens would be erased on the first pass, satisfying all other constraints and leaving a single token in the system.

5   Since high-level protocols can be built from combinations of lower-level protocols, coordinators can be hierarchically composed. A coordinator is a six-tuple (I; M; B; N; A; X) in which:

- I is a set of coordination interfaces.

- M is a set of modes.

10   - B is a set of bindings between interface elements (*e.g.*, control ports and message ports) and internal elements (*e.g.*, modes and triggers).

- N is a set of constraints between interface elements.

- A is a set of action triples for the coordinator.

- X is a set of subcoordinators.

15   Figs. 6A, 6B, 6C, and 6D show a few simple coordinators highlighting the bindings and constraints of the respective coordinators. With reference to Fig. 6A, a unidirectional data transfer coordinator 600 transfers data in one direction between two components (not shown) by connecting incoming receive message port 408 to outgoing receive message port 418 with a binding 602. With reference to Fig. 6B,

20   bidirectional data transfer coordinator 604 transfers data back and forth between two components (not shown) by connecting incoming receive message port 408 to outgoing receive message port 418 with binding 602 and connecting send message port 406 to incoming send message port 416 with a second binding 602. Unidirectional data transfer coordinator 600 and bidirectional data transfer

25   coordinator 604 simply move data from one message port to another. Thus each coordinator consists of bindings between corresponding ports on separate coordination interfaces.

With reference to Fig. 6C, state unification coordinator 606 ensures that a state port a 608 and a state port b 610 are always set to the same value. State

30   unification coordinator 606 connects state port a 608 to state port b 610 with binding

602. With reference to Fig. 6D, control state mutex coordinator 612 has a first constraint 618 and a second constraint 620 as follows:

(1) $c \Rightarrow \neg d$ and

5   (2) $d \Rightarrow \neg c$.

Constraints 618 and 620 can be restated as follows:

(1) A state port c 614 having a true value implies that a state port d 616 has a false value, and

10   (2) State port d 616 having a true value implies that state port c 614 has a false value.

A coordinator has two types of coordination interfaces: up interfaces that connect the coordinator to a second coordinator, which is at a higher level of design hierarchy and down interfaces that connect the coordinator either to a component or to

15   a third coordinator, which is at a lower level of design hierarchy. Down interfaces have names preceded with "˜". Round-robin coordinator 410 has six down coordination interfaces (previously referred to as coordinator coordination interface 412), with constraints that make the turning off of any coordinator control port 414 (also referred to as access control port) turn on the coordinator control port 414 of the

20   next coordinator coordination interface 412 in line. Table 2 presents all constituents of the round-robin coordinator.

Constituent	Value
coordination interfaces	$I = $ ˜AccessInterface$_{1\text{-}6}$
modes	$M = $ access$_{1\text{-}6}$
bindings	$B = \forall_{1 \le i \le 6}(\text{˜AccessInterface}_i.access,\ access_i)\ \cup$
constraints	$N = \forall_{1 \le i \le 6}(\forall_{(1 \le j \le 6) \wedge (i \ne j)}\ access_i \Rightarrow \neg access_j)$

actions	$A = \forall_{1 \leq i \leq 6} \text{access}_i : -\text{access}_i : +\text{access}_{(i+1)} \bmod 6$
subcoordinators	$X = \varnothing$

This tuple describes an implementation of a round-robin coordination protocol for a particular system with six components, as shown in round-robin coordinator 410. We use a coordination class to describe a general coordination protocol that may not have a fixed number of coordinator coordination interfaces. The coordination class is a six-tuple (Ic; Mc; Bc; Nc; Ac; Xc) in which:

- Ic is a set of coordination interface descriptors in which each descriptor provides a type of coordination interface and specifies the number of such interfaces allowed within the coordination class.

- Mc is a set of abstract modes that supplies appropriate modes when a coordination class is instantiated with a fixed number of coordinator coordination interfaces.

- Bc is a set of abstract bindings that forms appropriate bindings between elements when the coordination class is instantiated.

- Nc is a set of abstract constraints that ensures appropriate constraints between coordination interface elements are in place as specified at instantiation.

- Ac is a set of abstract action triples for the coordinator.

- Xc is a set of coordination classes (hierarchy).

While a coordinator describes coordination protocol for a particular application, it requires many aspects, such as the number of coordination interfaces and datatypes, to be fixed. Coordination classes describe protocols across many applications. The use of the coordination interface descriptors instead of coordination interfaces lets coordination classes keep the number of interfaces and datatypes undetermined until a particular coordinator is instantiated. For example, a round-robin coordinator contains a fixed number of coordinator coordination interfaces with specific bindings and constraints between the message and state ports on the fixed number of coordinator coordination interfaces. A round-robin

coordination class contains descriptors for the coordinator coordination interface type, without stating how many coordinator coordination interfaces, and instructions for building bindings and constraints between ports on the coordinator coordination interfaces when a particular round-robin coordinator is created.

5   5.   Components

A component is a six-tuple (I; A; M; V; S; X) in which:

- I is a set of coordination interfaces.

- A is a set of action triples.

- M is a set of modes.

10 - V is a set of typed variables.

- S is a set of subcomponents.

- X is a set of coordinators used to connect the subcomponents to each other and to the coordination interfaces.

Actions within a coordinator are fairly regular, and hence a large number of 15 actions can be described with a few simple expressions. However, actions within a component are frequently diverse and can require distinct definitions for each individual action. Typically a component's action triples are represented with a table that has three columns: one for the mode, one for the trigger, and one for the action code. Table 3 shows some example actions from a component that can use round-20 robin coordination.

Mode	Trigger	Action
access	tick	AccessInterface.s.send("Test message"); -access;
¬ access	tick	waitCount++;

A component resembles a coordinator in several ways (for example, the modes and coordination interfaces in each are virtually the same). Components can have 25 internal coordinators, and because of the internal coordinators, components do not always require either bindings or constraints. In the following subsections, various

aspects of components are described in greater detail. Theses aspects of components include variable scope, action transparency, and execution semantics for systems of actions.

A. Variable Scope

5 To enhance a component's modularity, all variables accessed by an action within the component are either local to the action, local to the immediate parent component of the action, or accessed by the immediate parent component of the action via state ports in one of the parent component's coordination interfaces. For a component's variables to be available to a hierarchical child component, they must be

10 exported by the component and then imported by the child of the component.

B. Action Transparency

An action within a component can be either a transparent action or an opaque action. Transparent and opaque actions each have different invocation semantics. The internal properties, *i.e.* control structures, variable, changes in state, operators,

15 etc., of transparent actions are visible to all coordination-centric design tools. The design tools can separate, observe, and analyze all the internal properties of opaque actions. Opaque actions are source code. Opaque actions must be executed directly, and looking at the internal properties of opaque actions can be accomplished only through traditional, source-level debugging techniques. An opaque action must

20 explicitly declare any mode changes and coordination interfaces that the opaque action may directly affect.

C. Action Execution

An action is triggered by an event, such as data arriving or departing a message port, or changes in value being applied to a state port. An action can change

25 the value of a state port, generate an event, and provide a way for the software system to interact with low-level device drivers. Since actions typically produce events, a single trigger can be propagated through a sequence of actions.

6.   Protocols Implemented with Coordination Classes

In this section, we describe several coordinators that individually implement some common protocols: subsumption, barrier synchronization, rendezvous, and dedicated RPC.

5   A.   Subsumption Protocol

A subsumption protocol is a priority-based, preemptive resource allocation protocol commonly used in building small, autonomous robots, in which the shared resource is the robot itself.

Fig. 7 shows a set of coordination interfaces and a coordinator for

10   implementing the subsumption protocol. With reference to Fig. 7, a subsumption coordinator 700 has a set of subsumption coordinator coordination interfaces 702, which have a subsume arbitrated coordinator control port 704 and an incoming subsume message port 706. Each subsume component 708 has a subsume component coordination interface 710. Subsume component coordination interface 710 has a

15   subsume arbitrated component control port 712 and an outgoing subsume message port 714. Subsumption coordinator 700 and each subsume component 708 are connected by their respective coordination interfaces, 702 and 710. Each subsumption coordinator coordination interface 702 in subsumption coordinator 700 is associated with a priority. Each subsume component 708 has a behavior that can be

20   applied to a robot (not shown). At any time, any subsume component 708 can attempt to assert its behavior on the robot. The asserted behavior coming from the subsume component 708 connected to the subsumption coordinator coordination interface 702 with the highest priority is the asserted behavior that will actually be performed by the robot. Subsume components 708 need not know anything about other components in

25   the system. In fact, each subsume component 708 is designed to perform independently of whether their asserted behavior is performed or ignored.

Subsumption coordinator 700 further has a slave coordinator coordination interface 716, which has an outgoing slave message port 718. Outgoing slave message port 718 is connected to an incoming slave message port 720. Incoming

30   slave message port 720 is part of a slave coordination interface 722, which is

connected to a slave 730. When a subsume component 708 asserts a behavior and that component has the highest priority, subsumption coordinator 700 will control slave 730 (which typically controls the robot) based on the asserted behavior.

The following constraint describes the basis of the subsumption coordinator

5   700's behavior:

$$\text{subsume}_p \;\Rightarrow\; \bigwedge_{i=1}^{p-1} \neg\text{subsume}_i$$

10

This means that if any subsume component 708 has a subsume arbitrated component control port 712 that has a value of true, then all lower-priority subsume arbitrated component control ports 712 are set to false. An important difference between round-robin and subsumption is that in round-robin, the resource access right

15   is transferred only when surrendered. Therefore, round-robin coordination has cooperative release semantics. However, in subsumption coordination, a subsume component 708 tries to obtain the resource whenever it needs to and succeeds only when it has higher priority than any other subsume component 708 that needs the resource at the same time. A lower-priority subsume component 708 already using

20   the resource must surrender the resource whenever a higher-priority subsume component 708 tries to access the resource. Subsumption coordination uses preemptive release semantics, whereby each subsume component 708 must always be prepared to relinquish the resource.

Table 4 presents the complete tuple for the subsumption coordinator.

Constituent	Value
coordination interfaces	$I = (\text{Subsume}_{1-n}) \cup (\text{Output})$
modes	$M = \text{subsume}_{1-n}$
bindings	$B = \forall_{1 \leq i \leq n} (\text{Subsume}_i.\text{subsume}, \text{subsume}_i) \cup$
constraints	$N = \forall_{1 \leq i \leq n} (\forall_{(1 \leq j \leq i)} \text{ subsume}_i \Rightarrow \neg\text{subsume}_j)$
actions	$A = \varnothing$
subcoordinators	$X = \varnothing$

B.   Barrier Synchronization Protocol

5   Other simple types of coordination that components might engage in enforce synchronization of activities. An example is barrier synchronization, in which each component reaches a synchronization point independently and waits. Fig. 8 depicts a barrier synchronization coordinator 800. With reference to Fig. 8, barrier synchronization coordinator 800 has a set of barrier synchronization coordination interfaces 802, each of which has a coordinator arbitrated state port 804, named wait.

10   Coordinator arbitrated state port 804 is connected to a component arbitrated state port 806, which is part of a component coordination interface 808. Component coordination interface 808 is connected to a component 810. When all components 810 reach their respective synchronization points, they are all released from waiting.

15   The actions for a barrier synchronization coordinator with n interfaces are:

$$\bigwedge_{0 \leq i < n} wait_i \quad : \quad : \quad \forall_{0 \leq j < n} - wait_j$$

In other words, when all wait modes (not shown) become active, each one is released. The blank between the two colons indicates that the trigger event is the guard condition becoming true.

C. Rendezvous Protocol

5 A resource allocation protocol similar to barrier synchronization is called rendezvous. Fig. 9 depicts a rendezvous coordinator 900 in accordance with the present invention. With reference to Fig. 9, rendezvous coordinator 900 has a rendezvous coordination interface 902, which has a rendezvous arbitrated state port 904. A set of rendezvous components 906, each of which may perform different

10 functions or have vastly different actions and modes, has a rendezvous component coordination interface 908, which includes a component arbitrated state port 910. Rendezvous components 906 connect to rendezvous coordinator 900 through their respective coordination interfaces, 908 and 902. Rendezvous coordinator 900 further has a rendezvous resource coordination interface 912, which has a rendezvous

15 resource arbitrated state port 914, also called available. A resource 916 has a resource coordination interface 918, which has a resource arbitrated state port 920. Resource 916 is connected to rendezvous coordinator 900 by their complementary coordination interfaces, 918 and 912 respectively.

With rendezvous-style coordination, there are two types of participants:

20 resource 916 and several resource users, here rendezvous components 916. When resource 916 is available, it activates its resource arbitrated state port 920, also referred to as its available control port. If there are any waiting rendezvous components 916, one will be matched with the resource; both participants are then released. This differs from subsumption and round-robin in that resource 916 plays

25 an active role in the protocol by activating its available control port 920.

The actions for rendezvous coordinator 900 are:

$$available_i \wedge wait_j : \quad : -available_i, -wait_j$$

This could also be accompanied by other modes that indicate the status after the rendezvous. With rendezvous coordination, it is important that only one component at a time be released from wait mode.

D.   Dedicated RPC Protocol

5   A coordination class that differs from those described above is dedicated RPC. Fig. 10 depicts a dedicated RPC system. With reference to Fig. 10, a dedicated RPC coordinator 1000 has an RPC server coordination interface 1002, which includes an RPC server imported state port 1004, an RPC server output message port 1006, and an RPC server input message port 1008. Dedicated RPC coordinator 1000 is

10   connected to a server 1010. Server 1010 has a server coordination interface 1012, which has a server exported state port 1014, a server input data port 1016, and a server output data port 1018. Dedicated RPC coordinator 1000 is connected to server 1010 through their complementary coordination interfaces, 1002 and 1012 respectively. Dedicated RPC coordinator 1000 further has an RPC client coordination

15   interface 1020, which includes an RPC client imported state port 1022, an RPC client input message port 1024, and an RPC client output message port 1026. Dedicated RPC coordinator 1000 is connected to a client 1028 by connecting RPC client coordination interface 1020 to a complementary client coordination interface 1030. Client coordination interface 1030 has a client exported state port 1032, a client

20   output message port 1034, and a client input message port 1036.

The dedicated RPC protocol has a client/server protocol in which server 1010 is dedicated to a single client, in this case client 1028. Unlike the resource allocation protocol examples, the temporal behavior of this protocol is the most important factor in defining it. The following transaction listing describes this temporal behavior:

25   Client 1028 enters blocked mode by changing the value stored at client exported state port 1032 to true.

Client 1028 transmits an argument data message to server 1010 via client output message port 1034.

Server 1010 receives the argument (labeled "a") data message via server input data port 1016 and enters serving mode by changing the value stored in server exported state port 1014 to true.

Server 1010 computes return value.

5   Server 1010 transmits a return (labeled "r") message to client 1020 via server output data port 1018 and exits serving mode by changing the value stored in server exported state port 1014 to false.

Client 1028 receives the return data message via client input message port 1036 and exits blocked mode by changing the value stored at client exported state port

10   1032 to false.

This can be presented more concisely with an expression describing causal relationships:

$$T_{RPC} = +client.blocked \rightarrow client.transmits \rightarrow$$

15     $$+server.serving \rightarrow server.transmits \rightarrow$$

$$(-server.serving \parallel client.receives) \rightarrow -client.blocked$$

The transactions above describe what is supposed to happen. Other properties of this protocol must be described with temporal logic predicates.

20

$$server.serving \Rightarrow client.blocked$$

$$server.serving \Rightarrow F(server.r.output)$$

$$server.a.input \Rightarrow F(server.serving)$$

25   The r in $server.r.output$ refers to the server output data port 1018, also labeled as the r event port on the server, and the a in $serving.a.input$ refers to server input data port 1016, also labeled as the a port on the server (see Fig. 10).

Together, these predicates indicate that (1) it is an error for server 1010 to be in serving mode if client 1028 is not blocked; (2) after server 1010 enters serving mode, a response message is sent or else an error occurs; and (3) server 1010 receiving a message means that server 1010 must enter serving mode. Relationships

5   between control state and data paths must also be considered, such as:

$$(client.a \Rightarrow client.blocked)$$

In other words, client 1028 must be in blocked mode whenever it sends an argument message.

10   The first predicate takes the same form as a constraint; however, since dedicated RPC coordinator 1000 only imports the client:blocked and server:serving modes (*i.e.,* through RPC client imported state port 1022 and RPC server imported state port 1004 respectively), dedicated RPC coordinator 1000 is not allowed to alter these values to comply. In fact, none of these predicates is explicitly enforced by a

15   runtime system. However, the last two can be used as requirements and guarantees for interface type-checking.

7.   System-Level Execution

Coordination-centric design methodology lets system specifications be executed directly, according to the semantics described above. When components and

20   coordinators are composed into higher-order structures, however, it becomes essential to consider hazards that can affect system behavior. Examples include conflicting constraints, in which local resolution semantics may either leave the system in an inconsistent state or make it cycle forever, and conflicting actions that undo one another's behavior. In the remainder of this section, the effect of composition issues

25   on system-level executions is explained.

A.   System Control Configurations

A configuration is the combined control state of a system—basically, the set of active modes at a point in time. In other words, a configuration in coordination-centric design is a bit vector containing one bit for each mode in the

30   system. The bit representing a control state is true when the control state is active

and false when the control state is inactive. Configurations representing the complete system control state facilitate reasoning on system properties and enable several forms of static analysis of system behavior.

B.   Action-Trigger Propagation

5   Triggers are formal parameters for events. As mentioned earlier, there are two types of triggers: (1) control triggers, invoked by control events such as mode change requests, and (2) data flow triggers, invoked by data events such as message arrivals or departures. Components and coordinators can both request mode changes (on the modes visible to them) and generate new messages (on the message ports visible to them). Using actions, these events can be propagated through the

10   components and coordinators in the system, causing a cascade of data transmissions and mode change requests, some of which can cancel other requests. When the requests, and secondary requests implied by them, are all propagated through the system, any requests that have not been canceled are confirmed and made part of the

15   system's new configuration.

Triggers can be immediately propagated through their respective actions or delayed by a scheduling step. Recall that component actions can be either transparent or opaque. Transparent actions typically propagate their triggers immediately, although it is not absolutely necessary that they do so. Opaque actions typically must

20   always delay propagation.

1.   Immediate Propagation

Some triggers must be immediately propagated through actions, but only on certain types of transparent actions. Immediate propagation can often involve static precomputation of the effect of changes, which means that certain actions may never

25   actually be performed. For example, consider a system with a coordinator that has an action that activates mode A and a coordinator with an action that deactivates mode B whenever A is activated. Static analysis can be used to determine in advance that any event that activates A will also deactivate B; therefore, this effect can be executed immediately without actually propagating it through A.

2. Delayed Propagation

Trigger propagation through opaque actions must typically be delayed, since the system cannot look into opaque actions to precompute their results. Propagation may be delayed for other reasons, such as system efficiency. For example, immediate

5   propagation requires tight synchronization among software components. If functionality is spread among a number of architectural components, immediate propagation is impractical.

C.   A Protocol Implemented with a Compound Coordinator

Multiple coordinators are typically needed in the design of a system. The

10   multiple coordinators can be used together for a single, unified behavior. Unfortunately, one coordinator may interfere with another's behavior.

Fig. 11 shows a combined coordinator 1100 with both preemption and round-robin coordination for controlling access to a resource, as discussed above. With reference to Fig. 11, components 1102, 1104, 1106, 1108, and 1110 primarily use

15   round-robin coordination, and each includes a component coordination interface 1112, which has a component arbitrated control port 1114 and a component output message port 1116. However, when a preemptor component 1120 needs the resource, preemptor component 1120 is allowed to grab the resource immediately. Preemptor component 1120 has a preemptor component coordination interface 1122. Preemptor

20   component coordination interface 1122 has a preemptor arbitrated state port 1124, a preemptor output message port 1126, and a preemptor input message port 1128.

All component coordination interfaces 1112 and preemptor component coordination interface 1122 are connected to a complementary combined coordinator coordination interface 1130, which has a coordinator arbitrated state port 1132, a

25   coordinator input message port 1134, and a coordinator output message port 1136. Combined coordinator 1100 is a hierarchical coordinator and internally has a round-robin coordinator (not shown) and a preemption coordinator (not shown). Combined coordinator coordination interface 1130 is connected to a coordination interface to round-robin 1138 and a coordination interface to preempt 1140.

30   Coordinator arbitrated state port 1132 is bound to both a token arbitrated control port

1142, which is part of coordination interface to round-robin 1138, and to a preempt

arbitrated control port 1144, which is part of coordination interface to preempt 1140.

Coordinator input message port 1134 is bound to an interface to a round-robin output

message port 1146, and coordinator output message port 1136 is bound to an interface

5   to round-robin input message port 1148.

Thus preemption interferes with the normal round-robin ordering of access to

the resource. After a preemption-based access, the resource moves to the component

that in round-robin-ordered access would be the successor to preemptor component

1120. If the resource is preempted too frequently, some components may starve.

10   D.   Mixing Control and Data in Coordinators

Since triggers can be control-based, data-based, or both, and actions can

produce both control and data events, control and dataflow aspects of a system are

coupled through actions. Through combinations of actions, designers can effectively

employ modal data flow, in which relative schedules are switched on and off based on

15   the system configuration.

Relative scheduling is a form of coordination. Recognizing this and

understanding how it affects a design can allow a powerful class of optimizations.

Many data-centric systems (or subsystems) use conjunctive firing, which means that a

component buffers messages until a firing rule is matched. When matching occurs,

20   the component fires, consuming the messages in its buffer that caused it to fire and

generating a message or messages of its own. Synchronous data flow systems are

those in which all components have only firing rules with constant message

consumption and generation.

Fig. 12A shows a system in which a component N1 1200 is connected to a

25   component N3 1202 by a data transfer coordinator 1204 and a component N2 1206 is

connected to component N3 1202 by a second data transfer coordinator 1208.

Component N3 1202 fires when it accumulates three messages on a port c 1210 and

two messages on a port d 1212. On firing, component N3 1202 produces two

messages on a port o 1214. Coordination control state tracks the logical buffer depth

for these components. This is shown with numbers representing the logical queue depth of each port in Fig. 12.

Fig. 12B shows the system of Fig. 12A in which data transfer coordinator 1204 and second data transfer coordinator 1208 have been merged to form a merged

5   data transfer coordinator 1216. Merging the coordinators in this example provides an efficient static schedule for component firing. Merged data transfer coordinator 1216 fires component N1 1200 three times and component N2 1206 twice. Merged data transfer coordinator 1216 then fires component N3 1202 twice (to consume all messages produced by component N1 1200 and component N2 1206).

10   Message rates can vary based on mode. For example, a component may consume two messages each time it fires in one mode and four each time it fires in a second mode. For a component like this, it is often possible to merge schedules on a configuration basis, in which each configuration has static consumption and production rates for all affected components.

15   E.   <u>Coordination Transformations</u>

In specifying complete systems, designers must often specify not only the coordination between two objects, but also the intermediate mechanism they must use to implement this coordination. While this intermediate mechanism can be as simple as shared memory, it can also be another coordinator; hence coordination may be, and

20   often is, layered. For example, RPC coordination often sits on top of a TCP/IP stack or on an IrDA stack, in which each layer coordinates with peer layers on other processing elements using unique coordination protocols. Here, each layer provides certain capabilities to the layer directly above it, and the upper layer must be implemented in terms of them.

25   In many cases, control and communication synthesis can be employed to automatically transform user-specified coordination to a selected set of standard protocols. Designers may have to manually produce transformations for nonstandard protocols.

F.   Dynamic Behavior with Compound Coordinators

Even in statically bound systems, components may need to interact in a fashion that appears dynamic. For example, RPC-style coordination often has multiple clients for individual servers. Here, there is no apparent connection between client and

5   server until one is forged for a transaction. After the connection is forged, however, the coordination proceeds in the same fashion as dedicated RPC.

Our approach to this is to treat the RPC server as a shared resource, requiring resource allocation protocols to control access. However, none of the resource allocation protocols described thus far would work efficiently under these

10   circumstances. In the following subsections, an appropriate protocol for treating the RPC as a shared resource will be described and how that protocol should be used as part of a complete multiclient RPC coordination class—one that uses the same RPC coordination interfaces described earlier—will be discussed.

1.   First Come/First Serve protocol (FCFS)

15   Fig. 13 illustrates a first come/first serve (FCFS) resource allocation protocol, which is a protocol that allocates a shared resource to the requester that has waited longest. With reference to Fig. 13, a FCFS component interface 1300 for this protocol has a request control port 1302, an access control port 1304 and a component outgoing message port 1306. A FCFS coordinator 1308 for this protocol has a set of

20   FCFS interfaces 1310 that are complementary to FCFS component interfaces 1300, having a FCFS coordinator request control port 1312, a FCFS coordinator access port 1314, and a FCFS coordinator input message port 1316. When a component 1318 needs to access a resource 1320, it asserts request control port 1302. When granted access, FCFS coordinator 1308 asserts the appropriate FCFS coordinator access port

25   1314, releasing FCFS coordinator request control port 1312.

To do this, FCFS coordinator 1308 uses a rendezvous coordinator and two round-robin coordinators. One round-robin coordinator maintains a list of empty slots in which a component may be enqueued, and the other round-robin coordinator maintains a list showing the next component to be granted access. When an FCFS

30   coordinator request control port 1312 becomes active, FCFS coordinator 1308 begins

a rendezvous access to a binder action. When activated, this action maps the appropriate component 1318 to a position in the round-robin queues. A separate action cycles through one of the queues and selects the next component to access the server. As much as possible, FCFS coordinator 1308 attempts to grant access to

5   resource 1320 to the earliest component 1318 having requested resource 1320, with concurrent requests determined based on the order in the rendezvous coordinator of the respective components 1318.

<div align="center">2.   Multiclient RPC</div>

Fig. 14 depicts a multiclient RPC coordinator 1400 formed by combining

10   FCFS coordinator 1308 with dedicated RPC coordinator 1000. With reference to Fig. 14, a set of clients 1402 have a set of client coordination interfaces 1030, as shown in Fig. 10. In addition, multiclient RPC coordinator 1400 has a set of RPC client coordination interfaces 1020, as shown in Fig. 10. For each RPC client coordination interface 1020, RPC client input message port 1024, of RPC client coordination

15   interface 1020, is bound to the component outgoing message port 1306 of FCFS coordinator 1308. Message transfer action 1403 serves to transfer messages between RPC client input message port 1024 and component outgoing message port 1306. For coordinating the actions of multiple clients 1402, multiclient RPC coordinator 1400 must negotiate accesses to a server 1404 and keep track of the values returned by

20   server 1404.

<div align="center">F.   Monitor Modes and Continuations</div>

Features such as blocking behavior and exceptions can be implemented in the coordination-centric design methodology with the aid of monitor modes. Monitor modes are modes that exclude all but a selected set of actions called continuations,

25   which are actions that continue a behavior started by another action.

<div align="center">1.   Blocking Behavior</div>

With blocking behavior, one action releases control while entering a monitor mode, and a continuation resumes execution after the anticipated response event. Monitor mode entry must be immediate (at least locally), so that no unexpected

30   actions can execute before they are blocked by such a mode.

Each monitor mode has a list of actions that cannot be executed when it is entered. The allowed (unlisted) actions are either irrelevant or are continuations of the action that caused entry into this mode. There are other conditions, as well. This mode requires an exception action if forced to exit. However, this exception action is not executed if the monitor mode is turned off locally.

When components are distributed over a number of processing elements, it is not practical to assume complete synchronization of the control state. In fact, there are a number of synchronization options available as detailed in Chou, P "Control Composition and Synthesis of Distributed Real-Time Embedded Systems", Ph.D. dissertation, University of Washington, 1998.

2. Exception Handling

Exception actions are a type of continuation. When in a monitor mode, exception actions respond to unexpected events or events that signal error conditions. For example, multiclient RPC coordinator 1400 can bind $\neg client.blocked$ to a monitor mode and set an exception action on $+server.serving$. This will signal an error whenever the server begins to work when the client is not blocked for a response.

8. A Complete System Example

Figure 15 depicts a large-scale example system under the coordination-centric design methodology. With reference to Fig. 15, the large scale system is a bimodal digital cellular network 1500. Network 1500 is for the most part a simplified version of a GSM (global system for mobile communications) cellular network. This example shows in greater detail how the parts of coordination-centric design work together and demonstrates a practical application of the methodology. Network 1500 has two different types of cells, a surface cell 1502 (also referred to as a base station 1502) and a satellite cell 1504. These cells are not only differentiated by physical position, but by the technologies they use to share network 1500. Satellite cells 1504 use a code division multiple access (CDMA) technology, and surface cells 1502 use a time division multiple access (TDMA) technology. Typically, there are seven frequency bands reserved for TDMA and one band reserved for CDMA. The goal is for as

much communication as possible to be conducted through the smaller TDMA cells, here surface cells 1502, because power requirements for a CDMA cells, here satellite cell 1504, increase with the number of users in the CDMA cell. Mobile units 1506, or wireless devices, can move between surface cells 1502, requiring horizontal handoffs between surface cells 1502. Several surface cells 1502 are typically connected to a switching center 1508. Switching center 1508 is typically connected to a telephone network or the Internet 1512. In addition to handoffs between surface cells 1502, the network must be able to hand off between switching centers 1508. When mobile units 1506 leave the TDMA region, they remain covered by satellite cells 1504 via vertical handoffs between cells. Since vertical handoffs require changing protocols as well as changing base stations and switching centers, they can be complicated in terms of control.

Numerous embedded systems comprise the overall system. For example, switching center 1508 and base stations, surface cells 1502, are required as part of the network infrastructure, but cellular phones, handheld Web browsers, and other mobile units 1506 may be supported for access through network 1500. This section concentrates on the software systems for two particular mobile units 1506: a simple digital cellular phone (shown in Fig. 16) and a handheld Web browser (shown in Fig. 24). These examples require a wide variety of coordinators and reusable components. Layered coordination is a feature in each system, because a function of many subsystems is to perform a layered protocol. Furthermore, this example displays how the hierarchically constructed components can be applied in a realistic system to help manage the complexity of the overall design.

To begin this discussion, we describe the cellular phone in detail, focusing on its functional components and the formalization of their interaction protocols. We then discuss the handheld Web browser in less detail but highlight the main ways in which its functionality and coordination differ from those of the cellular phone. In describing the cellular phone, we use a top-down approach to show how a coherent system organization is preserved, even at a high level. In describing the handheld

Web browser, we use a bottom-up approach to illustrate component reuse and bottom-up design.

A.   Cellular Phone

Fig. 16 shows a top-level coordination diagram of the behavior of a cell phone

5   1600. Rather than using a single coordinator that integrates the components under a single protocol, we use several coordinators in concert. Interactions between coordinators occur mainly within the components to which they connect.

With reference to Fig. 16, cell phone 1600 supports digital encoding of voice streams. Before it can be used, it must be authenticated with a home master switching

10   center (not shown). This authentication occurs through a registered master switch for each phone and an authentication number from the phone itself. There are various authentication statuses, such as full access, grey-listed, or blacklisted. For cell phone 1600, real-time performance is more important than reliability. A dropped packet is not retransmitted, and a late packet is dropped since its omission degrades the signal

15   less than its late incorporation.

Each component of cell phone 1600 is hierarchical. A GUI 1602 lets users enter phone numbers while displaying them and query an address book 1604 and a logs component 1606. Address book 1604 is a database that can map names to phone numbers and vice versa. GUI 1602 uses address book 1604 to help identify callers

20   and to look up phone numbers to be dialed. Logs 1606 track both incoming and outgoing calls as they are dialed. A voice component 1608 digitally encodes and decodes, and compresses and decompresses, an audio signal. A connection component 1610 multiplexes, transmits, receives, and demultiplexes the radio signal and separates out the voice stream and caller identification information.
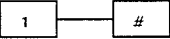
25   Coordination among the above components makes use of several of the coordinators discussed above. Between connection component 1610 and a clock 1612, and between logs 1606 and connection component 1610, are unidirectional data transfer coordinators 600 as described with reference to Fig. 6A. Between voice component 1608 and connection component 1610, and between GUI 1602 and

30   connection component 1610, are bidirectional data transfer coordinators 604, as

described with reference to Fig. 6B. Between clock 1612 and GUI 1602 is a state

unification coordinator 606, as described with reference to Fig. 6C. Between GUI

1602 and address book 1604 is a dedicated RPC coordinator 1000 as described with

reference to Fig. 10, in which address book 1604 has client 1028 and GUI 1602 has

5   server 1010.

There is also a custom GUI/log coordinator 1614 between logs 1606 and GUI

1602. GUI/log coordinator 1614 lets GUI 1602 transfer new logged information

through an r output message port 1616 on a GUI coordination interface 1618 to an r

input message port 1620 on a log coordination interface 1622. GUI/log coordinator

10   1614 also lets GUI 1602 choose current log entries through a pair of c output message

ports 1624 on GUI coordination interface 1618 and a pair of c input message ports

1626 on log coordination interface 1622. Logs 1606 continuously display one entry

each for incoming and outgoing calls.

1.   GUI Component

15   Fig. 17A is a detailed view of GUI component 1602, of Fig. 16. With

reference to Fig. 17A, GUI component 1602 has two inner components, a keypad

1700 and a text-based liquid crystal display 1702, as well as several functions of its

own (not shown). Each time a key press occurs, it triggers an action that interprets

the press, depending on the mode of the system. Numeric presses enter values into a

20   shared dialing buffer. When a complete number is entered, the contents of this buffer

are used to establish a new connection through connection component 1610. Table 5

shows the action triples for GUI 1602.

Mode	Trigger	Action
Idle	`1 ——— #`	numBuffer.append(keypress.val)
	Send	radio.send(numBuffer.val) +outgoingCall
	Disconnect	Nil
	Leftarrow	AddressBook.forward() +lookupMode
	Rightarrow	log.lastcall() +outlog
LookupMode	Leftarrow	AddressBook.forward()
	Rightarrow	AddressBook.backward()

An "Addr Coord" coordinator 1704 includes an address book mode (not shown) in which arrow key presses are transformed into RPC calls.

2. Logs Component

5   Fig. 17B is a detailed view of logs component 1606, which tracks all incoming and outgoing calls. With reference to Fig. 17B, both GUI component 1602 and connection component 1610 must communicate with logs component 1606 through specific message ports. Those specific message ports include a transmitted number message port 1720, a received number message port 1722, a change current received

10   message port 1724, a change current transmitted message port 1726, and two state ports 1728 and 1729 for presenting the current received and current transmitted values, respectively.

Logs component 1606 contains two identical single-log components: a send log 1730 for outgoing calls and a receive log 1740 for incoming calls. The interface

15   of logs component 1606 is connected to the individual log components by a pair of adapter coordinators, Adap1 1750 and Adap2 1752. Adap1 1750 has an adapter receive interface 1754, which has a receive imported state port 1756 and a receive output message port 1758. Adap1 1750 further has an adapter send interface 1760,

which has a send imported state port 1762 and a send output message port 1764.
Within Adap1, state port 1728 is bound to receive imported state port 1756, change
current received message port 1724 is bound to receive output message port 1758,
received number message port 1722 is bound to a received interface output message

5   port 1766 on a received number coordination interface 1768, change current
transmitted message port 1726 is bound to send output message port 1764, and state
port 1729 is bound to Up.rc is bound to send imported state port 1762 .

3.   Voice Component

Fig. 18A is a detailed view of voice component 1608 of Fig. 16. Voice

10   component 1608 has a compression component 1800 for compressing digitized voice
signals before transmission, a decompression component 1802 for decompressing
received digitized voice signals, and interfaces 1804 and 1806 to analog transducers
(not shown) for digitizing sound to be transmitted and for converting received
transmissions into sound. Voice component 1608 is a pure data flow component

15   containing sound generator 1808 which functions as a white-noise generator, a ring
tone generator, and which has a separate port for each on sound generator interface
1810, and voice compression functionality in the form of compression component
1800 and decompression component 1802.

4.   Connection Component

20   Fig. 18B is a detailed view of connection component 1610 of Fig. 16. With
reference to Fig. 18B, connection component 1610 coordinates with voice component
1608, logs component 1606, clock 1612, and GUI 1602. In addition, connection
component 1610 is responsible for coordinating the behavior of cell phone 1600 with
a base station that owns the surface cell 1502 (shown in Fig. 15), a switching center

25   1508 (shown in Fig. 15), and all other phones (not shown) within surface cell 1502.
Connection component 1610 must authenticate users, establish connections, and
perform handoffs as needed—including appropriate changes in any low-level protocols
(such as a switch from TDMA to CDMA).

Fig. 19 depicts a set of communication layers between connection component

30   1610 of cell phone 1600 and base station 1502 or switching center 1508. With

reference to Fig. 19, has several subcomponents, or lower-level components, each of which coordinates with an equivalent, or peer, layer on either base station 1502 or switching center 1508. The subcomponents of connection component 1610 include a cell phone call manager 1900, a cell phone mobility manager 1902, a cell phone radio

5 resource manager 1904, a cell phone link protocol manager 1906, and a cell phone transport manager 1908 which is responsible for coordinating access to and transferring data through the shared airwaves TDMA and CDMA coordination. Each subcomponent will be described in detail including how each fits into the complete system.

10 Base station 1502 has a call management coordinator 1910, a mobility management coordinator 1912, a radio resource coordinator 1914 (BSSMAP 1915), a link protocol coordinator 1916 (SCCO 1917), and a transport coordinator 1918 (MTP 1919). Switching center 1508 has a switching center call manager 1920, a switching center mobility manager 1922, (a BSSMAP 1924, a SCCP 1926, and an MTP 1928).

15 a.   Call Management

Fig. 20 is a detailed view of a call management layer 2000 consisting of cell phone call manager 1900, which is connected to switching center call manager 1920 by call management coordinator 1910. With reference to Fig. 20, call management layer 2000 coordinates the connection between cell phone 1600 and switching center

20 1508. Call management layer 2000 is responsible for dialing, paging, and talking. Call management layer 2000 is always present in cell phone 1600, though not necessarily in Internet appliances (discussed later). Cell phone call manager 1900 includes a set of modes (not shown) for call management coordination that consists of the following modes:

25   •   Standby

•   Dialing

•   RingingRemote

•   Ringing

•   CallInProgress

Cell phone call manager 1900 has a cell phone call manager interface 2002. Cell phone call manager interface 2002 has a port corresponding to each of the above modes. The standby mode is bound to a standby exported state port 2010. The dialing mode is bound to a dialing exported state port 2012. The RingingRemote

5 mode is bound to a RingingRemote imported state port 2014. The Ringing mode is bound to a ringing imported state port 2016. The CallInProgress mode is bound to a CallInProgress arbitrated state port 2018.

Switching center call manager 1920 includes the following modes (not shown) for call management coordination at the switching center:

10 • Dialing

• RingingRemote

• Paging

• CallInProgress

Switching center call manager 1920 has a switching center call manager coordination

15 interface 2040, which includes a port for each of the above modes within switching center call manager 1920.

When cell phone 1600 requests a connection, switching center 1508 creates a new switching center call manager and establishes a call management coordinator 1910 between cell phone 1600 and switching center call manager 1920.

20 b. Mobility Management

A mobility management layer authenticates mobile unit 1506 or cell phone 1600. When there is a surface cell 1502 available, mobility manager 1902 contacts the switching center 1508 for surface cell 1502 and transfers a mobile unit identifier (not shown) for mobile unit 1506 to switching center 1508. Switching center 1508

25 then looks up a home motor switching center for mobile unit 1506 and establishes a set of permissions assigned to mobile unit 1506. This layer also acts as a conduit for the call management layer. In addition, the mobility management layer performs handoffs between base stations 1502 and switching centers 1508 based on information received from the radio resource layer.

c.   Radio Resource

In the radio resource layer, radio resource manager 1904, chooses the target base station 1502 and tracks changes in frequencies, time slices, and CDMA codes. Cell phones may negotiate with up to 16 base stations simultaneously. This layer also

5   identifies when handoffs are necessary.

d.   Link Protocol

The link layer manages a connection between cell phone 1600 and base station 1502. In this layer, link protocol manager 1906 packages data for transfer to base station 1502 from cell phone 1600.

10   e.   Transport

Fig. 21A is a detailed view of transport component 1908 of connection component 1610. Transport component 1908 has two subcomponents, a receive component 2100 for receiving data and a transmit component 2102 for transmitting data. Each of these subcomponents has two parallel data paths a CDMA path 2104

15   and a TDMA/FDMA path 2106 for communicating in the respective network protocols.

Fig. 21B is a detailed view of a CDMA modulator 2150, which implements a synchronous data flow data path. CDMA modulator 2150 takes the dot-product of an incoming data signal along path 2152 and a stored modulation code for cell phone

20   1600 along path 2154, which is a sequence of chips, which are measured time signals having a value of −1 or +1.

Transport component 1908 uses CDMA and TDMA technologies to coordinate access to a resource shared among several cell phones 1600, *i.e.*, the airwaves. Transport components 1908 supersede the FDMA technologies (*e.g.*, AM and FM)

25   used for analog cellular phones and for radio and television broadcasts. In FDMA, a signal is encoded for transmission by modulating it with a carrier frequency. A signal is decoded by demodulation after being passed through a band pass filter to remove other carrier frequencies. Each base station 1502 has a set of frequencies—chosen to minimize interference between adjacent cells. (The area covered by a cell may be

30   much smaller than the net range of the transmitters within it.)

TDMA, on the other hand, coordinates access to the airwaves through time slicing. Cell phone 1600 on the network is assigned a small time slice, during which it has exclusive access to the media. Outside of the small time slice, cell phone 1600 must remain silent. Decoding is performed by filtering out all signals outside of the

5   small time slice. The control for this access must be distributed. As such, each component involved must be synchronized to observe the start and end of the small time slice at the same instant.

Most TDMA systems also employ FDMA, so that instead of sharing a single frequency channel, cell phones 1600 share several channels. The band allocated to

10   TDMA is broken into frequency channels, each with a carrier frequency and a reasonable separation between channels. Thus user channels for the most common implementations of TDMA can be represented as a two-dimensional array, in which the rows represent frequency channels and the columns represent time slices.

CDMA is based on vector arithmetic. In a sense, CDMA performs inter-cell-

15   phone coordination using data flow. Instead of breaking up the band into frequency channels and time slicing these, CDMA regards the entire band as an n-dimensional vector space. Each channel is a code that represents a basis vector in this space. Bits in the signal are represented as either 1 or –1, and the modulation is the inner product of this signal and a basis vector of mobile unit 1506 or cell phone 1600. This process

20   is called spreading, since it effectively takes a narrowband signal and converts it into a broadband signal.

Demultiplexing is simply a matter of taking the dot-product of the received signal with the appropriate basis vector, obtaining the original 1 or –1. With fast computation and the appropriate codes or basis vectors, the signal can be modulated

25   without a carrier frequency. If this is not the case, a carrier and analog techniques can be used to fill in where computation fails. If a carrier is used, however, all units use the same carrier in all cells.

Fig. 22 shows TDMA and CDMA signals for four cell phones 1600. With reference to Fig. 22, for TDMA, each cell phone 1600 is assigned a time slice during

30   which it can transmit. Cell phone 1 is assigned time slice t0, cell phone 2 is assigned

time slice t1, cell phone 3 is assigned time slice t2, and cell phone 4 is assigned time slice t3. For CDMA, each cell phone 1600 is assigned a basis vector that it multiplies with its signal. Cell phone 1 is assigned the vector:

5

$$\begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \end{pmatrix}$$

Cell phone 2 is assigned the vector:

$$\begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

10

Cell phone 3 is assigned the vector:

$$\begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix}$$

Cell phone 4 is assigned the vector:

$$\begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}$$

5

Notice that these vectors form an orthogonal basis.

B.   Handheld Web Browser

In the previous subsection, we demonstrated our methodology on a cell phone

with a top-down design approach. In this subsection, we demonstrate our

10   methodology with a bottom-up approach in building a handheld Web browser.

Fig. 23A is a LCD touch screen component 2300 for a Web browser GUI

(shown in Fig. 24A) for a wireless device 1506. With reference to Fig. 23A, a LCD

touch screen component 2300, has an LCD screen 2302 and a touch pad 2304.

Fig. 23B is a Web page access component 2350 for fetching and formatting

15   web pages. With reference to Fig. 23B, web access component 2350 has a page fetch

subcomponent 2352 and a page format subcomponent 2354. Web access component

2350 reads hypertext markup language (HTML) from a connection interface 2356,

sends word placement requests to a display interface 2358, and sends image requests

to the connection interface 2356. Web access component 2350 also has a character

20   input interface to allow users to enter page requests directly and to fill out forms on

pages that have forms.

Fig. 24A shows a completed handheld Web browser GUI 2400. With

reference to Fig. 24A, handheld Web browser GUI 2400, has LCD touch screen

component 2300, web access component 2350, and a pen stroke recognition

25   component 2402 that translates pen strokes entered on touch pad 2304 into characters.

Fig. 24B shows the complete component view of a handheld Web browser 2450. With reference to Fig. 24B, handheld Web browser 2450 is formed by connecting handheld Web browser GUI 2400 to connection component 1610 of cell phone 1600 (described with reference to Fig. 16) with bi-directional data transfer

5   coordinator 604 (described with reference to Fig. 6B). Handheld Web browser 2450 is an example of mobile unit 1506, and connects to the Internet through the cellular infrastructure described above. However, handheld Web browser 2450 has different access requirements than does cell phone 1600. For handheld Web browser 2450, reliability is more important than real-time delivery. Dropped packets usually require

10   retransmission, so it is better to deliver a packet late than to drop it. Real-time issues primarily affect download time and are therefore secondary. Despite this, handheld Web browser 2450 must coordinate media access with cell phones 1600, and so it must use the same protocol as cell phones 1600 to connect to the network. For that reason, handheld Web browser 2450 can reuse connection component 1610 from cell

15   phone 1600.

Debugging Techniques

In concept, debugging is a simple process. A designer locates the cause of undesired behavior in a system and fixes the cause. In practice, debugging—even of sequential software—remains difficult. Embedded systems are considerably more

20   complicated to debug than sequential software, due to factors such as concurrence, distributed architectures, and real-time concerns. Issues taken for granted in sequential software, like a schedule that determines the order of all events (the program), are nonexistent in a typical distributed system. Locating and fixing bugs in these complex systems requires many factors, including an understanding of the

25   thought processes underpinning the design.

Prior art research into debugging distributed systems is diverse and eclectic and lacks any standard notations. This application uses a standardized notation both to describe the prior art and the present invention. As a result of this standardized notation, the principles in the prior art follow those published in the referenced

30   works. However, the specific notation, theorems, etc., may differ.

The two general classes of debugging techniques are event-based debugging and state-based debugging. Most debugging techniques for general-purpose distributed systems are event based. Event-based debugging techniques operate by collecting event traces from individual system components and causally relating those

5   event traces. These techniques require an ability to determine efficiently the causal ordering among any given pair of events. Determining the causal order can be difficult and costly.

Events may be primitive, or they may be hierarchical clusters of other events. Primitive events are abstractions of individual local occurrences that might be

10 important to a debugger. Examples of primitive events in sequential programs are variable assignments and subroutine entries or returns. Primitive events for distributed systems include message send and receive events.

State-based debugging techniques are less commonly used in debugging distributed systems. State-based debugging techniques typically operate by presenting

15 designers with views or snapshots of a process state. Distributed systems are not tightly synchronized, and so these techniques traditionally involve only the state of individual processes. However, state-based debugging techniques can be applied more generally by relaxing the concept of an "instant in time" so that it can be effectively applied to asynchronous processes.

20 1.   Event-Based Debugging

In this section, prior art systems for finding and tracking meaningful event orderings, despite limits in observation, are described. Typical ways in which event orderings are used in visualization tools through automated space/time diagrams are

25 then described.

A. Event Order Determination and Observation

The behavior of a software system is determined by the events that occur and the order in which they occur. For sequential systems, this seems almost too trivial to mention; of course, a given set of events, such as

$$\{x := 2, \, x := x * 2, \, x := 5, \, y := x\},$$

arranged in two different ways may describe two completely different behaviors. However, since a sequential program is essentially a complete schedule of events, ordering is explicit. Sequential debugging tools depend on the invariance of this event schedule to let programmers reproduce failures by simply using the same inputs. In distributed systems, as in any concurrent system, it is neither practical nor efficient to completely schedule all events. Concurrent systems typically must be designed with flexible event ordering.

Determining the order in which events occur in a distributed system is subject to the limits of observation. An observation is an event record collected by an observer. An observer is an entity that watches the progress of an execution and records events but does not interfere with the system. To determine the order in which two events occur, an observer must measure them both against a common reference.

Fig. 25 shows a typical space/time diagram 2500, with space represented on a vertical axis 2502 and time represented on a horizontal axis 2504. With reference to Fig. 25, space/time diagram 2500 provides a starting point for discussing executions in distributed systems. Space/time diagram 2500 gives us a visual representation for discussing event ordering and for comparing various styles of observation. A set of horizontal world lines 2506, 2508, and 2510 each represent an entity that is stationary in space. The entities represented by horizontal world lines 2506, 2508, and 2510 are called processes and typically represent software processes in the subject system. The

entities can also represent any entity that generates events in a sequential fashion. The spatial separation in the diagram, along vertical axis 2502, represents a virtual space, since several processes might execute on the same physical hardware. A diagonal world line 2512 is called a message and represents discrete communications that pass between two processes. A sphere 2514 represents an event. In subsequent figures vertical axis 2502 and horizontal axis 2504 are omitted from any space/time diagrams, unless vertical axis 2502 and horizontal axis 2504 provide additional clarity to a particular figure.

Fig. 26 shows a space/time diagram 2600 of two different observations of a single system execution, taken by a first observer 2602 and a second observer 2604. With reference to Fig. 26, first observer 2602 and second observer 2604 are entities that record event occurrence. First observer 2602 and second observer 2604 must each receive distinct notifications of each event that occurs and each must record the events in some total order. First observer 2602 and second observer 2604 are represented in space/time diagram 2600 as additional processes, or horizontal world lines. Each event recorded requires a signal from its respective process to both first observer 2602 and second observer 2604. The signals from an event x 2606 on a process 2608 to both first observer 2602 and second observer 2604 are embodied in messages 2610 and 2612, respectively. First observer 2602 records event x 2606 as preceding an event y 2614. However, second observer 2604 records event y 2614 as preceding event x 2606. Such effects may be caused by nonuniform latencies within the system.

However, the observations of first observer 2602 and second observer 2604 are not equally valid. A valid observation is typically an observation that preserves the order of events that depend on each other. Second observer 2604 records the receipt of a message 2616 before that message is transmitted. Thus the observation from second observer 2604 is not valid.

Fig. 27 shows a space/time diagram 2700 for a special, ideal observer, called the real-time observer (RTO) 2702. With reference to Fig. 27, RTO 2702 can view each event immediately as it occurs. Due to the limitations of physical clocks, and

efficiency issues in employing them, it is usually not practical to implement RTO

2702. However, RTO 2702 represents an upper bound on precision in event-order

determination.

Fig. 28 shows a space/time graph 2800 showing two valid observations of a

5   system taken by two separate observers: RTO 2702 and a third observer 2802. With

reference to Fig. 28, there is nothing special about the ordering of the observation

taken by RTO 2702. Events d 2804, e 2806, and f 2808 are all independent events in

this execution. Therefore, the observation produced by RTO 2702 and the

observation produced by third observer 2802 can each be used to reproduce equivalent

10   executions of the system. Any observation in which event dependencies are preserved

is typically equal in value to an observation by RTO 2702. However, real-time

distributed systems may need additional processes to emulate timing constraints.

Fig. 29 is a space/time diagram 2900 of a methodological observer, called the

discrete Lamport Observer (DLO) 2902, that records each event in a set of ordered

15   bins. With reference to Fig. 29, DLO 2902 records an event 2904 in an ordered bin

2906 based on the following rule: each event is recorded in the leftmost bin that

follows all events on which it depends. DLO 2902 views events discretely and does

not need a clock. DLO 2902 does, however, require explicit knowledge of event

dependency. To determine the bin in which each event must be placed, DLO 2902

20   needs to know the bins of the immediately preceding events. The observation

produced by DLO 2902 is also referred to as a topological sort of the system

execution's event graph.

In the following, E is the set of all events in an execution. The immediate

predecessor relation, $\leadsto \subseteq E \times E$, includes all pairs ($e_a$, e_b) such that:

25   a) If e_a and e_b are on the same process, e_a precedes e_b with no intermediate

events.

b) If e_b is a receive event, e_a is the send event that generated the message.

Given these conditions, e_a is called the immediate predecessor of e_b.

Each event has at most two immediate predecessors. Therefore, DLO 2902 need only find the bins of at most two records before each placement. The transitive closure of the immediate predecessor relation forms a causal relation. The causal relation, $\leadsto \subseteq E \times E$, is the smallest transitive relation such that $e_i \rightarrow e_j \Rightarrow \ \leadsto e_j$.

5   This relation defines a partial order of events and further limits the definition of a valid observation. A valid observation is an ordered record of events from a given execution, *i.e.*, (R, $\prec$), where $e \in E \Rightarrow (record(e)) \in R$ and $\prec$ is an ordering operator. A valid observation has:

10      $e_i; e_j \in E$, $e_i \leadsto e_j \Rightarrow record\ (e_i) \prec record\ (e_j)$

     The dual of the causal relation is a concurrence relation. The concurrence relation, $E \times E$, includes all pairs (e_a, e_b) such that neither $e_a \leadsto e_b$ nor $e_b \leadsto e_a$. While the causal relation is transitive, the concurrence relation is not. The concurrence

15 relation is symmetric, while the causal relation is not.

B. *Event-Order Tracking*

     Debugging typically requires an understanding of the order in which events occur. Above, observers were presented as separate processes. While that treatment

20 simplified the discussion of observers it is typically not a practical implementation of an observer. When the observer is implemented as a physical process, the signals to indicate events would have to be transformed into physical messages and the system would have to be synchronized to enable all messages to arrive in a valid order.

     Fig. 30 depicts a space/time graph 3000 with each event having a label 3002.

25 With reference to Fig. 30, DLO 2902 can accurately place event records in their proper bins—even if received out of order—as long as it knows the bins of the immediate predecessors. If we know the bins in which events are recorded, we can determine something about their causality. Fortunately, it is easy to label each event

with the number of its intended bin. Labels 3002 are analogous to time and are typically called Lamport timestamps.

A Lamport timestamp is an integer t associated with an event e_i such that

5
$$e_i \rightsquigarrow e_j \Rightarrow t\,(e_i) < t(e_j)$$

Lamport timestamps can be assigned as needed, provided the labels of an event's immediate predecessors are known. This information can be maintained with a local counter, called a Lamport clock (not shown), t_{P_i}, on each process, P_i. The clock's

10   value is transmitted with each message M_j as t_{M_j}. Clock value t_{P_i} is updated with each event, as follows:

$$tpi = \begin{cases} \max\,(tMj, tpi) + 1; & \text{if } e \text{ is a receive event} \\ tpi + 1 & ; otherwise \end{cases}$$

15   A labeling mechanism is said to characterize the causal relation if, based on their labels alone, it can be determined whether two events are causal or concurrent. Although Lamport timestamps are consistent with causality (if $t(e_i) \geq t(e_j)$, then e_i $\rightsquigarrow e_j$), they do not characterize the causal relation.

Fig. 31 is a space/time graph 3100 that demonstrates the inability of scalar
20   timestamps to characterize causality between events. With reference to Fig. 31, space/time graph 3100 shows event e_1 3102, event e_2 3104, and event e_3 3106.

e_1 3102 causes e_2 3104, and also e_1 3102 is concurrent with e_3 3106 e_2 3104 is concurrent with e_3 3106 and it can be shown that e_3 3106 appears, when scalar timestamps are used, concurrent with both e_1 3102 and e_2 3104. However, since e_1
25   3102 $\rightsquigarrow e_2$ 3104 it is not possible for e_3 3106 to be concurrent with both.

66

Event causality can be tracked completely using explicit event dependence graphs, with directed edges from each event to its immediate predecessors. Unfortunately, this method cannot store enough information with each record to determine whether two arbitrarily chosen events are causally related without

5 traversing the dependence graph.

Other labeling techniques, such as vector timestamps, can characterize causality. The typical formulation of vector timestamps is based on the cardinality of event histories. A basis for vector timestamp is established based on the following definitions and theorems. An event history, $H(e_j)$, of an event e_j is the set of all

10 events, e_i, such that either since $e_i \leadsto e_j$ or $e_i \leadsto e_i = e_j$. The event history can be projected against specific processes. For a process P_j: the P_j history projection of $H(e_i)$, $H_{P_j}(e_i)$, is the intersection of $H(e_i)$ and the set of events local to P_j. The event graph represented by a space/time diagram can be partitioned into equivalence classes, with one class for each process. The set of events local to P_j is just the P_j

15 equivalence class.

The intersection of any two projections from the same process is identical to at least one of the two projections. Two history projections from a single process, $Hp(a)$ and $Hp(b)$, must satisfy one of the following:

20   a) $Hp(a) \subset Hp(b)$

     b) $Hp(a) = Hp(b)$

     c) $Hp(a) \supset Hp(b)$

The cardinality of $H_{P_j}(e_i)$ is thus the number of events local to P_j that causally precede

25 e_i and e_i itself. Since local events always occur in sequence, we can uniquely identify an event by its process and the cardinality of its local history.

For events e_a; e_b with $e_a \neq e_b$, $H_{Pea}(e_a) \subseteq H_{Pea}(e_b) \Rightarrow e_a \leadsto e_b$

Fig. 32 shows a space/time diagram 3200 with vector timestamped events. A vector timestamp 3202 is a vector label, t_e, assigned to each event, $e \in E$, such that the ith element represents $[H_{Pi}(e)]$. Given two events, e_1 and e_2, we can determine their causal ordering: if vector t_{ei} has a smaller value for its own process's entry than the other, t_{ej}, has at that same position, then $e_i \leadsto e_j$. If both vectors have larger values for their own process entries, then $e_i \parallel e_j$. It is not possible for both events to have smaller values for their own entries because for events e_a and e_b, $e_a \nrightarrow e_b$ implies $H_{Pea}(e_a) \supseteq H_{Pea}(e_b)$. It is not necessary to know the local processes of events to determine their causal order using vector timestamps.

The causal order of two vector timestamped events, e_a and e_b, from unknown processes can be determined with an element-by-element comparison of their vector timestamps:

$$\bigwedge_{i=1}^{n} t_{ea}[i] \leq t_{ea}[i] \;\Rightarrow\; e_a \rightarrow e_a$$

$$\neg \bigwedge_{i=1}^{n} t_{ea}[i] \leq t_{eb}[i] \;\wedge$$

$$\neg \bigwedge_{i=1}^{n} t_{eb}[i] \leq t_{ea}[i] \;\Rightarrow\; e_a \parallel e_b$$

Thus vector timestamps both fully characterize causality and uniquely identify each event in an execution.

Computing vector timestamps at runtime is similar to Lamport timestamp computation. Each process (P_s) contains a vector clock $(\hat{t}_{Ps})$ with elements for every process in the system, where $\hat{t}_{Ps}[s]$ always equals the number of events local to P_s. Snapshots of this vector counter are used to label each event, and snapshots are transmitted with each message. The recipient of a message with a vector snapshot can

update its own vector counter ($\hat{t}$ Pr) by replacing it with sup($\hat{t}$ Ps, $\hat{t}$ Pr), the element-wise

maximum of $\hat{t}$ Ps and $\hat{t}$ Pr.

This technique places enough information with each message to determine message ordering. It is performed by comparing snapshots attached to each message.

5   However, transmission of entire snapshots is usually not practical, especially if the system contains a large number of processes.

Vector clocks can however be maintained without transmitting complete snapshots. A transmitting process, P_s, can send a list that includes only those vector clock values that have changed since its last message. A recipient, P_r, then compares

10   the change list to its current elements and updates those that are smaller. This requires each process to maintain several vectors: one for itself and one for each process to which it has sent messages. However, change lists do not contain enough information to independently track message order.

The expense of maintaining vector clocks can be a strong deterrent to

15   employing them. Unfortunately, no technique with smaller labels can characterize causality. It has been shown that the dimension of the causal relation for an N-process distributed execution is N, and hence N-element vectors are the smallest labels characterizing causality.

The problem results from concurrence, without which Lamport time would be

20   sufficient. Concurrence can be tracked with concurrency maps, where each event keeps track of all events with which it is concurrent. Since the maps characterize concurrency, adding Lamport time lets them also characterize causality (the concurrency information disambiguates the scalar time). Unfortunately, concurrency maps can only be constructed after-the-fact, since doing so requires an examination of

25   events from all processes.

In some situations, distinguishing between concurrency and causality is not a necessity, but merely a convenience. There are compact labeling techniques that allow better concurrency detection than Lamport time. One such technique uses

interval clocks, in which each event record is labeled with its own Lamport time and the Lamport time of its earliest successor. This label then represents a Lamport time interval, during which the corresponding event was the latest known by the process. This gives each event a wider region with which to detect concurrence (indicated by overlapping intervals).

5

In cases in which there is little or no cross-process causality (few messages), interval timestamps are not much better than Lamport timestamps. In cases with large numbers of messages, however, interval timestamps can yield better results.

C. *Space/Time Displays in Debugging Tools*

10

Space/time diagrams have typically proven useful in discussing event causality and concurrence. Space/time diagrams are also often employed as the user display in concurrent program debugging tools.

The Los Alamos parallel debugging system uses a text based *time-process* display, and Idd uses a graphic display. Both of these, however, rely on an accurate global real-time clock (impractical in most systems).

15

Fig. 33 shows a Partial Order Event Tracer (POET) display 3300. The POET system supports several different languages and run-time environments, including Hermes, a high-level interpreted language for distributed systems, and Java. With reference to Fig. 33, POET display 3300 distinguishes among several types of events by shapes, shading, and alignment of corresponding message lines.

20

A Distributed Program Debugger (DPD) is based on a Remote Execution Manager (REM) framework. The REM framework is a set of servers on interconnected Unix machines in which each server is a Unix user-level process. Processes executing in this framework can create and communicate with processes elsewhere in the network as if they were all on the same machine. DPD uses space/time displays for debugging communication only, and it relies on separate source-level debuggers for individual processes.

25

2. Abstraction in Event-Based Debugging

Simple space/time displays can be used to present programmers with a wealth of information about distributed executions. Typically, however, space/time diagrams are too abstract to be an ultimate debugging solution. Space/time diagrams show

5   high-level events and message traffic, but they do not support designer interaction with the source code. On the other hand, simple space/time diagrams may sometimes have too much detail. Space/time diagrams display each distinct low-level message that contributes to a high-level transaction without support for abstracting the transaction.

10   Fig. 34 is a space/time diagram 3400 having a first compound event 3402 and a second compound event 3404. With reference to Fig. 34, even though a pair of primitive events are either causally related or concurrent, first and second compound events 3402 and 3404, or any other pair of compound events, might be neither causally related nor concurrent. Abstraction is typically applied across two

15   dimensions—events and processes—to aid in the task of debugging distributed software. Event abstraction represents sequences of events as single entities. A group of events may occasionally have a specific semantic meaning that is difficult to recognize, much as streams of characters can have a meaning that is difficult to interpret without proper spacing and punctuation. Event abstraction can in some

20   circumstances complicate the relationships between events.

Event abstraction can be applied in one of three ways: filtering, clustering, and interpretation. With event filtering, a programmer describes event types that the debugger should ignore, which are then hidden from view. With clustering, the debugger collects a number of events and presents the group as a single event. With

25   interpretation, the debugger parses the event stream for event sequences with specific semantic meaning and presents them to a programmer.

Process abstraction is usually applied only as hierarchical clustering. The remainder of this section discusses these specific event and process abstraction approaches.

A. *Event Filtering and Clustering*

Event filtering and clustering are techniques used to hide events from a

designer and thereby reduce clutter. Event filters exclude selected events from being

5   tracked in event-based debugging techniques. In most cases, this filtering is implicit

and cannot be modified without changing the source code because the source code

being debugged is designed to report only certain events to the debugger. When

deployed, the code will report all such events to the tool. This approach is employed

in both DPD and POET, although some events may be filtered from the display at a

10   later time.

An event cluster is a group of events represented as a single event. The

placement of an event in a cluster is based on simple parameters, such as virtual time

bounds and process groups. Event clusters can have causal ambiguities. For

example, one cluster may contain events that causally precede events in a second

15   cluster, while other events causally follow certain events in the second cluster.

Fig. 35 shows a POET display 3500 involving a first convex event cluster

3502 and a second convex event cluster 3504. POET uses a virtual-time-based

clustering technique that represents convex event clusters as single abstract events. A

convex event cluster is a set of event instances, C, such that for events

20

$$a, b, c \in E \text{ with } a, c \in C, a \leadsto b \wedge b \leadsto c \Rightarrow b \in C$$

Convex event clusters, unlike generic event clusters, cannot overlap.

B. *Event Interpretation (Specific Background for Behavioral*

25   *Abstraction)*

The third technique for applying event abstraction is interpretation, also

referred to as behavioral abstraction. Both terms describe techniques that use

debugging tools to interpret the behavior represented by sequences of events and

30   present results to a designer. Most approaches to behavioral abstraction let a designer

describe sequences of events using expressions, and the tools recognize the sequence of events through a combination of customized finite automata followed by explicit checks. Typically, matched expressions generate new events.

1. Event Description Language (EDL)

One of the earliest behavioral abstraction technique was event description language (EDL), in which event streams are pattern-matched using shuffle automata. A match produces a new event that can, in turn, be part of another pattern. Essentially, abstract events are hierarchical and are built from the bottom up.

This approach can recognize event patterns that contain concurrent events. There are, however, several weaknesses in this approach. First, shuffle automata match events from a linear stream, which is subject to a strong observational bias. In addition, even if the stream constitutes a valid observation, interleaving may cause false intermediates between an event and its immediate successor. Finally, concurrent events appear to occur in some specific order.

Bates partially compensates for these problems in three ways. First, all intermediates between two recognized events are ignored—hence, false intermediates are skipped. Unfortunately, true intermediates are also skipped, making error detection difficult. Second, the shuffle operator, Δ, is used to identify matches with concurrent events. Unfortunately, shuffle recognizes events that occur in any order, regardless of whether they are truly ordered in the corresponding execution. For example, $e_1 \Delta e_2$ can match with either $e_1 \prec e_2$ or $e_2 \prec e_1$ in the event stream, but this means the actual matches could be: $e_1 \rightsquigarrow e_2$, $e_2 \rightsquigarrow e_1$, in addition to the $e_1 \parallel e_2$ that the programmer intended to match. Third, the programmer can prescribe explicit checks to be performed on each match before asserting the results. However, the checks allowed do not include causality or concurrence checks.

2. Chain Expressions

Chain expressions, used in the Ariadne parallel debugger are an alternate way to describe distributed behavior patterns that have both causality and concurrence. These behavioral descriptions are based on chains of events (abstract sequences not

bound to processes), p-chains (chains bound to processes), and pt-chains (composed p-chains). The syntax for describing chain expressions is fairly simple, with <a b> representing two causally related events and |[a b]| representing two concurrent events.

5   The recognition algorithm has two functions. First, the algorithm recognizes the appropriate event sequence from a linear stream, using a nondeterminate finite automaton (NFA). Second, the algorithm checks the relationships between specific events

For example, when looking for sequences that match the expression < |[a b]|

10   c> *(viz.,* a and b are concurrent, and both causally precede c), Ariadne will find the sequence a b c and then verify the relationships among them. Unfortunately, the fact that sequences are picked in order from a linear stream before relationships are checked can cause certain matches to be missed. For example, |[a b]| and |[b a]| should have the same meaning, but they do not cause identical matches. This is

15   because Ariadne uses NFAs as the first stage in event abstraction. In the totally ordered stream to which an NFA responds, either a will precede b, preventing the NFA for the second expression from recognizing the string, or b will precede a, preventing the NFA for the first expression from recognizing the string.

3.   Distributed Abstraction

20   The behavioral abstraction techniques described so far rely on centralized abstraction facilities. These facilities can be distributed, as well. The BEE (Basis for distributed Event Environments) project is a distributed, hierarchical, event-collection system, with debugging clients located with each process.

Fig. 36 show a Basis for distributed Event Environments (BEE) abstraction

25   facility 3600 for a single client. With reference to Fig. 36, event interpretation is performed at several levels. The first is an event sensor 3602, inserted into the source of the program under test and invoked whenever a primitive event occurs during execution. The next level is an event generator 3604, where information— including timestamps and process identifiers—is attached to each event. Event

generator 3604 uses an event table 3606 to determine whether events should be passed to an event handler 3608 or simply dropped. Event handler 3608 manages event table 3606 within event generator 3604. Event handler 3608 filters and collects events and routes them to appropriate event interpreters (not shown). Event interpreters (not shown) gather events from a number of clients (not shown) and aggregate them for presentation to a programmer. Clients and their related event interpreters are placed together in groups managed by an event manager (not shown). A weakness of this technique is that it does not specifically track causality. Instead, this technique relies on the real-timestamps attached to specific primitive or abstract events. However, as discussed above these timestamps are not able to characterize causality.

C.   *Process Clustering*

Most distributed computing environments feature flat process structures, with few formally stated relationships among processes. Automatic process clustering tools can partially reverse-engineer a hierarchical structure to help remove spurious information from a debugger's view. Intuitively, a good cluster hierarchy should reveal, at the top level, high-level system behavior, and the resolution should improve proportionally with the number of processes exposed. A poor cluster hierarchy would show very little at the top level and would require a programmer to descend several hierarchical levels before getting even a rough idea about system behavior. Process clustering tools attempt to identify common interaction patterns—such as client-server, master-slave, complex server, layered system, and so forth. When these patterns are identified, the participants are clustered together. Clusters can then serve as participants in interaction patterns to be further clustered. Typically, these cluster hierarchies are strictly trees, as shown in Fig. 37, which depicts a hierarchical construction of process clusters 3700. With reference to Fig. 37, a square node 3702 represents a process (not shown) and a round node 3704 represents a process cluster (not shown).

Programmers can choose a debugging focus, in which they specify the aspects and detail levels they want to use to observe an execution. With reference to Fig. 37, a representative debugging focus that includes nodes I, J, E, F, G, and H is shown.

One drawback of this approach is that when a parent cluster is in focus, none of its children can be. For example, if we wanted to look at process K in detail, we would also need to expose at least as much detail for processes E and L and process cluster D.

5   Each process usually participates in many types of interactions with other processes. Therefore, the abstraction tools must heuristically decide between several options. These decisions have a substantial impact on the quality of a cluster hierarchy. Prior art systems have evaluated the quality of a clustering tool by measuring the cohesion, which though expressed quantitatively is actually a qualitative

10   measurement (the higher the better) within a cluster and the coupling, a qualitative measure of the information clusters must know about each other (the higher the worse), between clusters. For a cluster P of m processes, cohesion is quantified by:

$$\text{Cohesion}(P) = \frac{\sum_{i<j}\text{Sim}_f(p_i, p_j)}{m(m-1)/2}$$

15

where $\text{Sim}_f(P_1, P_2)$ is a similarity metric that equals:

$$\text{Sim}_f = \frac{A\langle \hat{C}_{P_1} \mid \hat{C}_{P_2}\rangle}{\|\hat{C}_{P_1}\| \cdot \|\hat{C}_{P_2}\|}$$

20   Here, $\langle \hat{a} \mid \hat{b}\rangle$ denotes the scaler product of vectors $\hat{a}$ and $\hat{b}$, and $\|\hat{a}\|$ denotes the magnitude of vector $\hat{a}$. C_{P_1} and C_{P_2} are process characteristic vectors—in them, each element contains a value between 0 and 1 that indicates how strongly a particular characteristic manifests itself in each process. Characteristics can include keywords, type names, function references, etc. A is a value that equals 1 if any of the

25   following apply:

- P_1 and P_1 are instantiations of the same source.

- P_1 and P_2 are unique instantiations of their own source.

- P_1 and P_2 communicate with each other.

A equals 0 if none of these is true (*e.g.*, P_1 and P_2 are nonunique instantiations of separate source that do not communicate with each other). Coupling is quantified by:

$$\text{Coupling}(P) = \frac{\sum_{ij} Sim_f(p_i, q_j)}{mn}$$

where $q_j \in Q$, Q is the complement of P, and $n = |Q|$. The quality of a cluster is quantified as its Coupling minus its Cohesion. In many cases, these metrics match many of the characteristics that intuitively differentiate good and poor clusters, as shown in Figs. 38A, B, and C. With reference to Figs. 38A and C, Cohesion is high where clusters correspond to heavy communication and where clusters correspond to processes instantiated from the same source code. Coupling is shown to be low in each of the above cases. With reference to Fig. 38B, Coupling is high when clusters do not correspond to heavily communicating processes or to instances of the same source code. It is not clear, however, that the cluster in Fig. 38C should be assigned the same quality value as the cluster in Fig. 38A. Using these metrics, Kunz achieved qualities of between :15 and :31 for his clustering techniques. However, it is hard to tell what this means in terms of cluster usefulness.

3. State-Based Debugging

State-based debugging techniques focus on the state of the system and the state changes caused by events, rather than on events themselves. The familiar source-level debugger for sequential program debugging is state-based. This source-level debugger lets designers set breakpoints in the execution of a program, enabling them to investigate the state left by the execution to that point. This source-level debugger

also lets programmers step through a program's execution and view changes in state caused by each step.

Concurrent systems have no unique meaning for an instant in execution time. Stopping or single-stepping the whole system can unintentionally, but substantially, 5 change the nature of interactions between processes.

A.   *Consistent Cuts and Global State*

In distributed event-based debugging, the concept of causality is typically of such importance that little of value can be discussed without a firm understanding of 10 causality and its implications. In distributed state-based debugging, the concept of a global instant in time is equally important.

Here again, it may seem intuitive to consider real-time instants as the global instants of interest. However, just as determining the real-time order of events is not practical or even particularly useful, finding accurate real-time instants makes little 15 sense. Instead, a global instant is represented by a consistent cut. A consistent cut is a cut of an event dependency graph representing an execution that (a) intersects each process exactly once and (b) points all dependencies crossing the cut in the same direction. Like real-time instants, consistent cuts have both a past and a future. These are the subgraphs on each side of the cut.

20   Fig. 39 shows that consistent cuts can be represented as a jagged line across the space/time diagram that meets the above requirements. With reference to Fig. 39, a space/time graph 3900 is shown having a first cut 3902 and a second cut 3904. All events to the left of either first cut 3902 or second cut 3904 are in the past of each cut, and all events to the right are in the future of each cut, respectively. First cut 25 3902 is a consistent cut because no message travels from the future to the past. Second cut 3904, however, is not consistent because a message 3906 travels from the future to the past.

Figs. 40A, B, and C show that a distributed execution shown in a space/time diagram 4000 can be represented by a lattice of consistent cuts 4002, in which τ is

the start of the execution and $\perp$ is system termination. With reference to Figs. 40A, B, and C, lattice of consistent cuts 4002 represents the global statespace traversed by a single execution. Since lattice of consistent cuts 4002's size is on the order of $|E|^{|P|}$, it, unlike space/time diagrams, is never actually constructed. In the remainder

5   of this chapter, to describe properties of consistent cut lattices, the symbol $\xrightarrow{1}$ relates cuts such that one immediately precedes the other and $\sim$ relates cuts between which there is a path.

B.   *Single Stepping in a Distributed Environment*

10   Controlled stepping, or single-stepping, through regions of an execution can help with an analysis of system behavior. The programmer can examine changes in state at the completion of each step to get a better understanding of system control flow. Coherent single-stepping for a distributed system requires steps to align with a path through a normal execution's consistent cut lattice.

15   DPD works with standard single-process debuggers (called client debuggers), such as DBX, GDB, etc. Programmers can use these tools to set source-level break-points and single-step through individual process executions. However, doing so leaves the other processes executing during each step, which can yield unrealistic executions.

20   Zernic gives a simple procedure for single-stepping using a post-mortem traversal of a consistent cut lattice. At each point in the step process, there are two disjoint sets of events: the past set, or events that have already been encountered by the stepping tool, and the future set, or those that have yet to be encountered. To perform a step, the debugger chooses an event, e_i, from the future such that any

25   events it depends on are already in the past, *i.e.,* there are no future events, e_f, such that $e_f \sim e_i$. This ensures that the step proceeds between two consistent cuts related by $\xrightarrow{1}$. The debugger moves this single event to the past, performing any necessary actions.

To allow more types of steps, POET's support for single-stepping uses three disjoint sets: executed, ready, and nonready. The executed set is identical to the past set in "Using Visualization Tools to Understand Concurrency," by D. Zernik, M. Snir, and D. Malki, *IEEE Software 9*, 3 (1992), pp. 87-92. The ready set contains all events that are fully enabled by events in the future, and the contents of the nonready set have some enabling events in either the ready or nonready sets. Using these sets, it is possible to perform three different types of steps: global-step, step-over, and step-in. Global-step and step-over may progress between two consistent cuts not related $\xrightarrow{\ l\ }$ (*i.e.,* there may be several intermediate cuts between the step cuts).

A global-step is performed by moving all events from the ready set into the past. Afterwards, the debugger must move to the ready set all events in the nonready set whose dependencies are in the executed set. A global-step is useful when the programmer wants information about a system execution without having to look at any process in detail.

The step-over procedure considers a local, or single-process, projection of the ready and nonready sets. To perform a step, it moves the earliest event from the local projections into the executed set and executes through events on the other processes until the next event in the projection is ready. This ensures that the process in focus will always have an event ready to execute in the step that follows.

Step-in is another type of local step. Unlike step-over, step-in does not advance the system at the completion of the step; instead, the system advance is considered to be a second step. Figs. 41A, B, C, and D show a space/time diagram before a step 4100 and a resulting space/time diagram after performing a global-step 4102, a step-over 4104, and a step-in 4106.

C.   *Runtime Consistent Cut Algorithms*

It is occasionally necessary to capture consistent cuts at runtime. To do so, each process performs some type of cut action (*e.g.,* state saving). This can be done with barrier synchronization, which erects a temporal barrier that no process can pass

until all processes arrive. Any cut taken immediately before, or immediately after, the barrier is consistent. However, with barrier synchronization, some processes may have a long wait before the final process arrives.

A more proactive technique is to use a process called the cut initiator to send
5   perform-cut messages to all other system processes. Upon receiving a perform-cut message, a process performs its cut action, sends a cut-finished message to the initiator, and then suspends itself. After the cut initiator receives cut-finished messages from all processes, it sends each of them a message to resume computation.

The cut obtained by this algorithm is consistent: no process is allowed to send
10   any messages from the time it performs its own cut action until all processes have completed the cut. This means that no post-cut messages can be received by processes that have yet to perform their own cut action. This algorithm has the undesirable characteristic of stopping the system for the duration of the cut. The following algorithms differ in that they allow some processing to continue.

15   1.   Chandy-Lamport Algorithm

The Chandy-Lamport algorithm does not require the system to be stopped. Once again, the cut starts when a cut initiator sends perform-cut messages to all of the processes. When a process receives a perform-cut message, it stops all work, performs its cut action, and then sends a mark on each of its outgoing channels; a
20   mark is a special message that tells its recipient to perform a cut action before reading the next message from the channel. When all marks have been sent, the process is free to continue computation. If the recipient has already performed the cut action when it receives a mark, it can continue working as normal.

Each cut request and each mark associated with a particular cut are labeled
25   with a cut identifier, such as the process ID of the cut initiator and an integer. This lets a process distinguish between marks for cuts it has already performed and marks

until all processes arrive. Any cut taken immediately before, or immediately after, the barrier is consistent. However, with barrier synchronization, some processes may have a long wait before the final process arrives.

A more proactive technique is to use a process called the cut initiator to send perform-cut messages to all other system processes. Upon receiving a perform-cut message, a process performs its cut action, sends a cut-finished message to the initiator, and then suspends itself. After the cut initiator receives cut-finished messages from all processes, it sends each of them a message to resume computation.

The cut obtained by this algorithm is consistent: no process is allowed to send any messages from the time it performs its own cut action until all processes have completed the cut. This means that no post-cut messages can be received by processes that have yet to perform their own cut action. This algorithm has the undesirable characteristic of stopping the system for the duration of the cut. The following algorithms differ in that they allow some processing to continue.

1. Chandy-Lamport Algorithm

The Chandy-Lamport algorithm does not require the system to be stopped. Once again, the cut starts when a cut initiator sends perform-cut messages to all of the processes. When a process receives a perform-cut message, it stops all work, performs its cut action, and then sends a mark on each of its outgoing channels; a mark is a special message that tells its recipient to perform a cut action before reading the next message from the channel. When all marks have been sent, the process is free to continue computation. If the recipient has already performed the cut action when it receives a mark, it can continue working as normal.

Each cut request and each mark associated with a particular cut are labeled with a cut identifier, such as the process ID of the cut initiator and an integer. This lets a process distinguish between marks for cuts it has already performed and marks for cuts it has yet to perform.

2.   Color-Based Algorithms

The Chandy-Lamport algorithm works only for FIFO (First In First Out) channels. If a channel is non-FIFO, a post-cut message may outrun the mark and be inconsistently received before the recipient is even aware of the cut, *i.e.*, it is received
5   in the cut's past. The remedy to this situation is a color-based algorithm. Two such algorithms are discussed below.

The first is called the two-color, or red-white, algorithm. With this algorithm, information about the cut state is transferred with each message. Each process in the system has a color. Processes not currently involved in a consistent cut are white,
10   and all messages transmitted are given a white tag. Again, there is a cut initiator that sends perform-cut messages to all system processes. When a process receives this request, it halts, performs the cut action, and changes its color to red. From this point on, all messages transmitted are tagged with red to inform the recipients that a cut has occurred.

15   Any process can accept a white message without consequence, but when a white process receives a red message, it must perform its cut action before accepting the message. Essentially, white processes treat red messages as cut requests. Red processes can accept red messages at any time, without consequence.

A disadvantage of the two-color algorithm is that the system must reset all of
20   the processes back to white after they have completed their cut action. After switching back, each process must treat red messages as if they were white until they are all flushed from the previous cut. After this, each process knows that the next red message it receives signals the next consistent cut.

This problem is addressed by the three-color algorithm, which resembles the
25   two-color algorithm in that every process changes color after performing a cut; it differs in that every change in color represents a cut. For colors zero through two, if a process with the color c receives a message with the color $(c-1) \bmod 3$, it registers this as a message-in-flight (see below). On the other hand, if it receives a message with the color $(c+1) \bmod 3$, it must perform its cut action and switch color

to $(c + 1)$ mod 3 before receiving the message. Of course, this can now be generalized to n-color algorithms, but three colors are usually sufficient.

Programmers may need to know about messages transmitted across the cut, or messages-in-flight. In the two-color algorithm, messages-in-flight are simply white

5   messages received by red processes. These can all be recorded locally, or the recipient can report them to the cut initiator. In the latter case, each red process simply sends the initiator a record of any white messages received.

It is not safe to switch from red to white in the two-color algorithm until the last message-in-flight has been received. This can be detected by associating a

10   counter with each process. A process increments its counter for each message sent and decrements it for each message received. When the value of this counter is sent to the initiator at the start of each process's cut action, the initiator can use the total value to determine the total number of messages-in-flight. The initiator simply decrements this count for each message-in-flight notification it receives.

15   **D.**   *State Recovery—Rollback and Replay*

Since distributed executions tend to be nondeterministic, it is often difficult to reproduce bugs that occur during individual executions. To do so, most distributed debuggers contain a rollback facility that returns the system to a previous state. For

20   this to be feasible, all processes in the system must occasionally save their state. This is called checkpointing the system. Checkpoints do not have to save the entire state of the system. It is sufficient to save only the changes since the last checkpoint. However, such incremental checkpointing can prolong recovery.

DPD makes use of the UNIX fork system call to perform checkpointing for

25   later rollback. When fork is called, it makes an exact copy of the calling process, including all current states. In the DPD checkpoint facility, the newly forked process is suspended and indexed. Rollback suspends the active process and resumes an indexed process. The problem with this approach is that it can quickly consume all system memory, especially if checkpointing occurs too frequently. DPD's solution is

to let the programmer choose the checkpoint frequency through use of a slider in its GUI.

Processes must sometimes be returned to states that were not specifically saved. In this case, the debugger must do additional work to advance the system to the desired point. This is typically called replay and is performed using event trace information to guide an execution of the system. In replay, the debugger chooses an enabled process *(i.e.,* one whose next event has no pending causal requirements) and executes it, using the event trace to determine where the process needs to block for a message that may have arrived asynchronously in the original execution. When the process blocks, the debugger chooses the next enabled process and continues from there. In this way, a replay is causally identical to the original execution.

Checkpoints must be used in a way that prevents domino effects. The domino effect occurs when rollbacks force processes to restore more than one state. Domino effects can roll the system back to the starting point. Fig. 42 shows a space time diagram 4200 for a system that is subject to the domino effect during rollback. With reference to Fig. 42, if the system requests a rollback to checkpoint c_3 4202 of process P_3 4204, all processes in the system must roll back to c_1 *(i.e.,* roll back to $P_3 . c_2$ 4206 requires a roll back to $P_2 . c_2$ 4208, which requires a roll back to $P_1 . c_2$ 4210, which requires a roll back to $P_3 . c_1$ 4212, which requires a roll back to $P_2 . c_1$ 4214, which requires a final roll back to $P_1 . c_1$ 4216). The problem is caused by causal overlaps between message transfers and checkpoints. Performing checkpoints only at consistent cuts avoids a domino effect.

E.   *Global State Predicates*

The ability to detect the truth value of predicates on global state yields much leverage when debugging distributed systems. This technique lets programmers raise flags when global assertions fail, set global breakpoints, and monitor interesting aspects of an execution. Global predicates are those whose truth value depends on the state maintained by several processes. They are typically denoted with the symbol Φ. Some examples include $\left(\sum_i c_i > 20\right)$ and $\left(c_1 < 20 \wedge c_2\, 5\right)$, where c_i is some variable in

process P_i that stores positive integers. In the worst case (such as when $\left(\sum_i c_i > 20\right)$ is false for an entire execution), it may be necessary to get the value of all such variables in all consistent cuts. In the following discussion, we use the notation C_a $| = \Phi$ to indicate that Φ is true in consistent cut C_a.

5   At this point, it is useful to introduce branching time temporal logic. Branching time temporal logic is predicate logic with temporal quantifiers, P, F, G, H, A, and E.   PΦ is true in the present if Φ was true at some point in the past; FΦ is true in the present if Φ will be true at some point in the future; GΦ is true in the present if Φ will be true at every moment in the future; and HΦ is true in the present if

10   Φ was true at every moment of the past. Notice that GΦ is the same as $\neg F \neg \Phi$, and HΦ is the same as $\neg P \neg \Phi$.

Since global time passage in distributed systems is marked by a partially ordered consistent cut lattice rather than by a totally ordered stream, we need the quantifiers A, which precedes a predicate that is true on all paths, and E, which

15   precedes a predicate that is true on at least one path. So, $AF\Phi$ is true in the consistent cut representing the present if Φ is true at least once on all paths in the lattice leaving this cut. $EP\Phi$ is true in the consistent cut representing the present if Φ is true on at least one path leading to this cut.

A monotonic global predicate is a predicate Φ such that $C_a | = \Phi \Rightarrow C_a | = AG \Phi$.

20   A monotonic global predicate is one that remains true after becoming true. An unstable global predicate, on the other hand, is a predicate Φ such that $C_a | = \Phi$ $\Rightarrow C_a | = EG \neg \Phi$. An unstable global predicate is one that may become false after becoming true.

1.   Detecting Monotonic Global Predicates

25   Monotonic predicates can be detected any time after becoming true. One algorithm is to occasionally take consistent cuts and evaluate the predicate at each. In fact, it is not necessary to use consistent cuts, since any transverse cut whose future is a subset of the future of the consistent cut in which the predicate first became true will also show the predicate true.

2. Detecting Unstable Global Predicates

Detecting arbitrary unstable global predicates can take at worst $|E|^{|P|}$ time, where $|E|^{|P|}$ is the size of an execution's consistent cut lattice, $[E]$ is the number of events in the execution, and $[P]$ is the number of processes. This is so, because it may be necessary to test for the predicate in every possible consistent cut. However, there are a few special circumstances that allow $|E|$ time algorithms.

Some unstable global predicates are true on only a few paths through the consistent cut lattice, while others are true on all paths. The prior art describes predicate qualifiers definitely Φ for predicates that are true on all paths *(i.e.,* $\top\,| = A\ F\ \Phi)$ and possibly Φ for those that are true on at least one path *(i.e.,* $\top\,| => E\ F\ \Phi)$.

The detection of possibly Φ for weak conjunctive predicates, or global predicates that can be expressed as conjunctions of local predicates, is $\varphi\,(|E|)$. The algorithm for this is to walk a path through the consistent cut lattice that aligns with a single process, P_t, until either (1) the process's component of Φ is true or (2) there is no way to proceed without diverging from P_t. In either case, the target process is switched and the walk continued. This algorithm continues until it reaches a state in which all components of the predicate are true or until it reaches $\perp$. In this way, if there are any consistent cuts where all parts of the predicate simultaneously hold, the algorithm will encounter at least one.

Detection of possibly Φ for weak disjunctive predicates, or global predicates that can be expressed as disjunctions of local predicates, is also $\varphi\,(|E|)$; it is the same algorithm as above, except it halts at the first node where *any* component is true. However, weak conjunctive and disjunctive predicates constitute only a small portion of the types of predicates that could be useful in debugging distributed systems.

4. Conclusions

Complicating the debugging of heterogenous embedded systems are designs composed of concurrent and distributed processes. Most of the difficulty in debugging distributed systems results from concurrent processes with globally

unscheduled and frequently asynchronous interactions. Multiple executions of a system can produce wildly varying results—even if they are based on identical inputs. The two main debugging approaches for these systems are event based and state based.

5   Event-based approaches are monitoring approaches. Events are presented to a designer in partially ordered event displays, called space/time displays. These are particularly good at showing inter-process communication over time. They can provide a designer with large amounts of information in a relatively small amount of space.

10   State-based approaches focus locally on the state of individual processes or globally on the state of the system. Designers can observe individual system states, set watches for specific global predicates, step through executions, and set breakpoints based on global state predicates. These approaches deal largely with snapshots, considering temporal aspects only as differences between snapshots.

15   As distributed systems increase in size and complexity, the sheer volume of events generated during an execution grows to a point where it is exceedingly difficult for designers to correctly identify aspects of the execution that may be relevant in locating a bug. For distributed system debugging techniques to scale to larger and faster systems, behavioral abstraction will typically become a necessity to help
20   designers identify and interpret complicated behavioral sequences in a system execution. Finally, embedded systems must execute in a separate environment from the one in which they were designed and embedded systems may also run for long periods of time without clear stopping points. Debugging them requires probes to report debugging information to a designer during the execution. These probes
25   inevitably alter system behavior, which can mask existing bugs or create new bugs that are not present in the uninstrumented system. While it is not possible to completely avoid these probe effects, they can be minimized through careful placement, or masked through permanent placement.

Static Control Graphs

A static control graph (SCG) is a graph-theoretic representation of all pure control constraints. Fig. 43 shows a simple SCG. It is a bi-partite digraph, having two types of nodes: conjunctive nodes 4300, which, as the name implies, produce results only when all incident edges 4302 are satisfied, and disjunctive nodes 4304, 4306, and 4308, which produce results if any incident edge 4302 is satisfied. Disjunctive nodes 4304, 4306, and 4308 correspond to modes 102 in components 100 and coordinators 410 (as previously shown in Fig. 1 and Fig. 4) throughout the system. An SCG for a complete system simultaneously represents all control constraints.

An SCG is a triple, $G = (C, D, E)$, in which:

- C is a set of conjunctive nodes 4300.

- D is a set of disjunctive nodes 4304, 4306, and 4308.

- $E \subseteq [\{T_f, T_t\} \times \{H_f; H_t\} \times ((C \times D) \cup (D \times C))]$ is a set of directed, labeled edges 4302. Edges are sensitive to either a false value or a true value at their tail 4310 (T_f or T_t) and enforce either a false value or a true value at their head 4312 (H_f or H_t). These are represented visually by a bubble at the appropriate end for a false value or the lack of a bubble for a true value.

An edge 4302 in an SCG can be either enabled or disabled; it produces the value true or the value false. Fig. 44 illustrates a graphic notation for edge labels. Edges 4400 and 4402 marked with a bubble on head 4312, as in Fig. 44A and Fig. 44B, assert the value false when activated. When there is no mark on head 4312, as in Fig. 44C and Fig. 44D, an edge 4404 and 4406 asserts the value true when activated. A bubble on tail 4310, as in Fig. 44B and Fig. 44D, indicates that edge 4406 and 4402 is sensitive to false on the node it exits. The lack of such bubbles, as in Fig. 44A and Fig. 44C, indicates that edge 4404 and 4400 is sensitive to true.

Referring back to Fig. 43, the figure shows a simple SCG in which a node d 4306 must be active whenever a conjunction (a ∧ b ∧ c) 4314 is active and inactive whenever a node e 4308 is active. Although this looks similar to a Boolean network, it differs because the SCG edges represent implication, not connection. This is

5   illustrated in Fig. 45, which shows a Boolean network OR node 4500; when all inputs and outputs are negated, it is equivalent to an AND node 4502 (by DeMorgan's). A disjunctive SCG node with all inputs and outputs negated 4504 is equivalent to a disjunctive node with no inputs and outputs negated 4506.

Each SCG has a Boolean characteristic function. This is a Boolean 164

10   function that is true for each configuration in which no constraints are violated and that is false for configurations with violated constraints. Fig. 46 shows two SCGs along with their reduced characteristic functions, in which the functions (*i.e.,* a ∧ b ⟹ ¬c) are reduced to functions using only basic operators, (*i.e.,* ¬(a ∧ b ∧ c)). Fig. 46A shows conjunction without negation, whereas Fig. 46B shows conjunction with

15   negation. Characteristic functions for SCGs with several conjunctive nodes are the conjunction of all constraints and, as such, may not be satisfiable.

Fig. 47 shows the impact of edge semantics on SCG. Edges that are incident upon conjunctive nodes (*i.e.,* edges that take the form (d, c) for d ∈ D, c ∈ C) are called sensing edges 4700 and 4702; edges incident upon disjunctive nodes (*i.e.,*

20   edges that take the form (c, d)) are called enforcing edges 4704 and 4706. All edges and nodes are labeled with their respective source object (*i.e.,* a mode/control port combination for a disjunctive node, and a constraint for a conjunctive node). When a bubble is placed at the head of an enforcing edge 4708, as in Fig. 47A, it has different semantics than when the bubble is placed at the tail of an enforcing edge 4710, as in

25   Fig. 45B; at the head of a sensing edge 4712, as in Fig. 45C; or at the tail of a sensing edge 4714, as in Fig. 45D. However, the latter three have identical semantics.

Activation influences for conjunctive nodes are always apparent in an SCG: the disjunctive nodes that appear at the tail side of incident edges. As a result,

30   conjunctive edges need never be labeled. Disjunctive nodes are frequently mapped to

modes in the system; hence, they may have hidden activation influences. In these circumstances, node labels must be applied to indicate the modes to which they are mapped. It may not always be possible for all edges to assert their value. Edges that are prevented from doing so are said to be violated.

5   *1.   Instability and Dynamic Properties*

Although SCGs embody static relationships between modes, through unstable configurations (*viz.*, configurations that are temporarily invalid and must hence attempt resolution), they can also model some dynamic properties. Recall the behavior of a rendezvous coordinator 4800; it contains interfaces 4802, 4804, 4806,

10   and 4808 for two types of components: resource users 4810 and resources 4812. Rendezvous coordinator 4800 lets resource users 4810 enter a waiting mode 4814, and resources 4812 enter an available mode 4816. When possible, rendezvous coordinator 4800 releases a waiting component 4818 and an available component 4820 together. This can be modeled by the SCG shown in Fig. 48B. The SCG has three

15   conjunctive nodes 4822, 4824, and 4826 that each sense (1) whether a particular wait node 4828, 4830, or 4832 and an avail node 4834 are simultaneously active and (2) whether there are any active wait nodes 4828, 4830, and 4832 with precedence. When one of conjunctive nodes 4822, 4824, or 4826 is satisfied, it releases its respective wait node 4828, 4830, or 4832 and avail node 4834—causing conjunctive

20   node 4822, 4824, or 4826 to cease being satisfied.

An important property of an SCG is the any stable state property (ASSP). ASSP of SCGs states that there is at least one stable configuration. An SCG is said to have this property if it has at least one configuration in which all constraints are simultaneously satisfied. A graph without the ASSP is shown in Fig. 49. Trying to

25   enforce all constraints results in an inactive node b 4900 forcing a node c 4902 to be inactive, which in turn forces node b 4900 to be active, which forces node c 4902 to be active—forcing node b 4900 to be inactive and restarting the cycle.

It can be shown that finding the ASSP is NP-Complete, meaning that there are probably no general, efficient algorithms for determining whether an SCG has this

30   property. A 3-SAT problem, which is a well-known NP-Complete problem,

commonly used to prove by polynomial time reduction that other computational problems are NP-Hard, can be reduced to ASSP in polynomial time. Fig. 50A shows the SCG for a 3-SAT problem. Fig. 50B shows the 3-SAT problem. Fig. 50C shows the representative characteristic functions for the 3-SAT problem.

5   Two enforcing edges can conflict and possibly cause race conditions in certain configurations. To avert this, edge labels also contain a priority to indicate which will be enforced if there is a conflict. Usually, these priorities are supplied by the coordinators that define the constraint and hence the enforcing edge. Although finding a complete constraint system solution is NP-Hard, it is possible to present a

10   solution consistent with a designer's expectations in less than exponential time.

2. *Petri-net Similarities*

SCGs are similar to Petri-nets in several significant ways. Petri-nets are also bi-partite digraphs with the node types - transitions and places. A system state is represented by a marking of places, where each place can be either marked by a token

15   or not marked at all. A transition fires if there is a token on each of the places on the opposite side of incoming edges. When a transition fires, it consumes all enabling tokens and places tokens on each of the places on the opposite side of outgoing edges.

      The main similarity between Petri-nets and SCGs is the conjunctive and disjunctive behaviors of transitions and places. The way in which a place in a Petri-

20   net becomes marked is similar to the way in which a disjunctive node in an SCG is changed—namely, incident edges manipulate the state directly. The main difference is that, with Petri-nets, the only way a place can become unmarked is if a transition on an outgoing edge removes the token. With SCGs, the only conjunctive nodes that can cause changes in state are those on the incoming edges.

25   ### 3. *Construction of SCGs*

SCG construction uses Boolean constraints from a standard product-of- sums. A standard form Boolean constraint is a tuple (I, O, R, A), in which:

-     I is a set of input literals.

-     O is a set of output literals.

- R is a set of disjunctions on input values, *i.e.,* $R \subseteq 2^I$.

- A is a set of output literals matched with conjunctions on values in R and values in $C \subseteq O \times 2^R$.

Algorithm 1 shows how an SCG is constructed from a set of constraints in a system.

5

Algorithm 1 - SCG construction

Require: *M*, a set of modes, and *S*, a set of boolean constraints in standard form.
    create $G = (C, D, E)$, a new control graph
    for all $m \in M$ **do**
10
        add **new** d_m to D // add a disjunctive node
    end for
    for all $s \in S$ **do**
        for all $a \in s.A$ **do**
            add **new** c_{sa} to C // add a conjunctive node
15
            for all $o \in a.O$ **do**
                add **new** edge (c_{sa}, o) to E
            end for
            for all $i \in s.I$ **do**
                add **new** edge (i, c_{sa}) to E
20
            end for
        end for
    end for

---

25   Algorithm 1 takes a set of modes and constraints, and for each mode generates a new disjunctive node, and for each constraint adds a new conjunctive node. For each left argument of the constraint, the algorithm adds a new edge from the disjunctive node that corresponds to the argument to the conjunctive node. For each right argument of the constraint, the algorithm adds a new edge from the conjunctive node to the
30   disjunctive node that corresponds to the argument.

4. *Constraint Conflict Detection*

The most important use of SCGs is in finding constraint enforcement conflicts. Constraint enforcement conflicts occur when two active constraints try to force a disjunctive node in opposite directions. Since constraints are prioritized, any conflicts
35   can be resolved at runtime in favor of the constraint with the highest priority. However, resolution in this fashion can cause unexpected, undesired behavior (*i.e.,*

bugs) in the embedded system. Pre-preemptive debugging, in the form of constraint conflict detection, aids designers in catching such problems.

There are two different types of constraint conflicts: first order and n^{th} order. First order conflicts are fairly straightforward to detect, but detecting n^{th} order

5   conflicts is, in general, as hard as detecting ASSP. A number of techniques can be used to reduce the complexity of nth order conflict detection. The first technique conservatively approximates conjunctive nodes as disjunctive nodes. This technique is polynomial and detects all possible conflicts, but it may also deliver a number of false positives. The second technique exploits the hierarchical properties of systems

10   designed using coordination-centric modeling to cache partial results; for certain SCGs called well-composed, this technique can deliver results in polynomial time. Many subgraphs in typical SCGs are replicated over and over. In these cases, replication can be exploited to deliver results in polynomial time. Finally, subgraphs may not relate to others, and in these cases, it is possible to analyze them separately.

15   ### A.   *First Order Conflicts*

First order conflicts occur when two potentially coactive enforcing edges have opposite effects on the same disjunctive node. Fig. 51A shows one such conflict: if nodes a, b, c, and d 5100, 5102, 5104, and 5106, respectively, are all simultaneously active, a conflict 5108 occurs between $c \wedge d$ 5110 and $a \wedge b$ 5112. Since the

20   conflicting edges 5114 and 5116 are prioritized, it is easy for the runtime system to dynamically resolve this in favor of $c \wedge d$ 5110.

First order conflicts are easy to detect—for each disjunctive node, simply find all enforcing edges that conflict in sense and trace back to their respective conjunctive nodes.

25   ### B.   *n^{th} Order Conflicts*

Fig. 51B shows a potential simple second order conflict between the terms $d \wedge g \wedge h$ 5118 and $a \wedge b$ 5120. In this case, priority labels 5122, 5124, and 5126 of the edges cause the $d \wedge g \wedge h$ 5118 term to be ignored whenever there is a conflict

between them. Cyclic n^{th} order conflicts are particularly bad, because they can cause instability in the underlying system—even with priority assignments.

Finding and eliminating n^{th} order conflicts is another NP-Hard problem, so we are reduced to practical techniques that seem to perform well given the characteristics of common SCGs. Experience has shown that these graphs frequently embody exponential control state spaces; therefore, it is essential to avoid enumerating the entire control state space in conflict detection.

Algorithm 2 shows one conflict detection algorithm that can usually attain reasonable performance. This algorithm produces a complete closure of the SCG, such that conflicts are identified by conflicts in conjunctive node output.

The conjunction between terms is represented by a single conjunctive node, and a potential conflict is realized if and only if all intermediate disjunctive nodes are satisfied. In practice, SCGs and the systems from which they are derived often have characteristics that can be exploited to find these potential conflicts efficiently.

Algorithm 2 - Flattening static control graphs

Require: (D, C, E) is a copy of a system's static control graph

  for all $d_i \in D$, (c_j, d_i) and $(d_i, c_k) \in E$ **do**

    if consistent $((c_j, d_k), (d_i, c_k))$ **then**

      if absent (c_j, c_k) **then**

        add **new** c_{jk} to C

        for all $(d_m; c_j) \in E$ **do**

          add **new** $(d_m; c_{jk})$ to E

        end for

        for all $(d_n; c_k) \in E$ with $d_n \neq d_i$ **do**

          add **new** (d_n, c_{jk}) to E

        end for

        for all $(c_k, d_p) \in E$ **do**

          add **new** (c_{jk}, d_p) to E

        end for

      end if

    end if

  end for

Algorithm 2 finds all disjunctive nodes and, for each of them, takes all consistent edges and creates a new conjunctive node for each. For each of the new conjunctive nodes, the algorithm creates new edges (1) for each edge on the conjunctive node, on the tail side of the left edge and (2) for each edge on the conjunctive node, on the

5   head side of the right edge. The consistency check ensures that no literals in one conjunctive node conflict with literals in the other (*e.g.*, it will return false if c_j contains a and c_k contains $\neg a$). The absence check ensures that the new node will not be redundant.

Fig. 52A shows a portion of an SCG before application of Algorithm 2

10   (flattening), and Fig. 52B shows the SCG after flattening. A new constraint between end nodes 5200, 5202, 5204, and 5206 is constructed; it includes c_i 5210 and edges 5212, 5214, and 5216 entering and leaving c_i 5210. This constraint represents the cascading effect through a node d_i 5208.

To find an upper bound on the space and time required, assume that each

15   possible consistent, distinct, conjunctive node will be created and that each of these will fan out to all disjunctive nodes. This means that for n disjunctive nodes, space for 3^n conjunctive nodes and $2n3^n$ edges (at most n edges on both sides of each conjunctive node) is required. Assuming that a vast majority of these are created during the execution of the algorithm, time must be approaching $4n3^n$. These space

20   and time requirements are extremely sensitive to the initial fan-in and fan-out of nodes in the graph, the initial number of conjunctive nodes, and the length of acyclic, self-consistent paths (*i.e.*, paths along which a single change effect can be propagated).

Once the graph has been flattened, true conflicts can be ascertained by tracing back pairs of edges that assert different values on any disjunctive node and by

25   determining whether their source conjunctive nodes are mutually exclusive. An example is shown in Fig. 53. Outgoing edges 5300 and 5302 from c_i 5304 and c_k 5306 form a potential conflict 5308. To determine whether a conflict exists, the literals that make up the respective conjunctions are compared, treating unused variables as "don't cares." As Table 6 shows, conflict 5308 occurs for the

30   configuration $d_i \wedge d_j \wedge \neg d_k \wedge d_l$.

95

Table 6. Finding the conflict from Fig. 54.

	d_i	d_j	d_k	d_l
c_l	1	1	0	--
c_k	--	--	0	1
equivalent	1	1	1	1
conflict	1	1	0	1

Algorithm 2 highlights instability, which shows up as simple cycles. For example, the instability in Fig. 49 is visible in Fig. 54 as four simple cycles 5400,

5   5402, 5404, and 5406 after flattening.

There are several techniques available to improve the performance of Algorithm 2 for a wide variety of SCGs. The following subsections describe some of them.

1.   *Disjunctive Graph Approximation*

10   A special case of SCGs is one in which each conjunctive edge has only a single input. In this case, Algorithm 2 becomes a variety of Warshall's algorithm, in which conjunctive nodes just define relationships between disjunctive nodes. If there are only a few conjunctive nodes with more than one incident edge, each of them can be replaced with several conjunctive nodes with single incident edges.

15   This approach yields a conservative approximation of the conflicting edges, with the possibility of a large number of false positives. If the number of false positives that concern the designer is small, they can be individually verified against the original graph.

2.   *Hierarchical Reduction*

20   Recall that one element of a component's coordination interface is a set of guarantees. Guarantees are summaries of component properties that are already verified. These guarantees can include summaries of internal relationships between control ports on the interface. Fig. 55A shows one such summary. Although the actual relationship between x 5500 and $\neg y$ 5502 involves a number of internal nodes

5504, 5506, 5508, and 5510, as seen in Fig. 55, none of them are part of the interface; therefore, they can be summarized as a single, independent node 5512, marked with "?", as shown in Fig. 55B.

Since our methodology encourages hierarchical composition, stable state consistency can often be applied hierarchically to these graphs. In this approach, each component provides a summary of the relationships between all interface modes. A system that is well-composed (as shown in Fig. 56) in this fashion will have the number of nodes visible in each scope less than some reasonable constant c, and the ratio of the number of non-interface nodes within a scope to the number of interface nodes is always greater than some constant factor d, where d > 1. With this, there are $\frac{n \log_d n}{c}$ different scopes to analyze, and the time required for each scope

is then less than or equal to $4c3^c$; thus the time required for an entire well composed graph is:

$$t \le \frac{4c3^c n \log_d n}{c}$$

which can be stated as t ≤ Pn lg n, where P is the constant $\frac{4(3^c)}{\lg^d}$. This means that hierarchical, stable-state consistency for well-composed graphs is O (n lg n).

While it is not possible to force arbitrary systems to fit this construct, it is possible to determine whether a system is well-composed before attempting this technique. Furthermore, for components supplied with models, the system can still use the interaction summary; therefore, it is not always necessary for components to be well-composed internally. Since the summary can be cached with components, the cost for preparing it can be amortized over many attempts at analysis.

3. *Exploiting Replication*

There are many protocols that have a large number of connected components, but how the components interact is independent of their number. In protocols such as token ring or subsumption, in which the interface is replicated with predictable

5   relationships between instances, it may not be necessary to verify all possible conflicts between all components. For example, with subsumption, components interact cleanly with the protocol, and conflicts between components are minimal. The conflicts between a component and a protocol can be treated as independent from the other components plugged into a protocol, so the component can be simply checked

10   against the protocol.

4. *Orthogonal Reduction*

Often, interactions between interfaces in a layered system are through opaque actions. Therefore, each layer can be considered a separate entity, and the graph for each layer can be constructed independently of the other graphs. In these cases,

15   flattening need not consider the system as a whole, but merely the subgraphs for each particular layer.

Dynamic Control Graphs

A dynamic control graph (DCG) includes pure control actions as well as pure control constraints. DCGs can be defined as a triple (Γ_d, D_d, E_e) wherein:

20   • Γ_d is a set of conjunctive and action nodes, *i.e.*,

$\Gamma_d \subseteq (C \cup A)$,

where C is the universe of all possible conjunctive nodes, and A is the universe of all possible actions.

• D_d is a set of disjunctive nodes.

25   • $E_e \subseteq [\{T_f, T_t\} \times \{H_f, H_t\} \times \{T, N\} \times ((\Gamma_d \times D_d) \cup (D_d \times \Gamma_d))]$

is a set of directed edges that are labeled either Transient (T) or Continuous (N).

Fig. 57 depicts a DCG 5700. With reference to Fig. 57, a pure control action (not shown) is a transparent action that is triggered only by a control transient and that

produces only a control transient. In DCG 5700, the pure control action is represented as a conjunctive node, an action node 5702, with outgoing dashed edges 5704 and 5706, and incoming dashed edge 5708. A pure control constraint (not shown) is one that constrains only modes (not shown). As described above, an action can be one of two types: an instantaneous action or a delayed action. The instantaneous action type is executed immediately when its trigger is received, and the delayed action type is executed at some point in time after its trigger is received. The delay of an action is embodied in its action node 5702.

Fig. 58 depicts a DCG 5800 with an action node 5802. With reference to Fig. 58, action node 5802 has no apparent trigger. With this, DCG 5800 fully characterizes all control aspects of the action (although the data aspect, which in this case is the trigger, is not included in any control graph). Fig. 59 shows a DCG 5900 for an action (not shown) that is transparent with respect to control interactions. Fig. 60 shows a DCG 6000 for an action (not shown) that is opaque with respect to control; the graph shows a conservative approximation of the action's control behavior.

As mentioned earlier, it is better to model coordinator transitions through explicit actions rather than through static control graph (SCG) instability. Figs. 61A and 61B are two DCGs 6100 and 6102, for rendezvous coordinator 900, of Fig. 9, and a rendezvous coordinator with two-participant preemption, respectively. With reference to Figs. 61A and 61B, there are n control actions (not shown) (one for each component (not shown) with a wait coordination interface (not shown)). Each of the n control actions is represented in DCGs 6100 and 6102 by an action node 6104, and each action node is guarded by the conjunction of all wait modes 6106 with lower precedence being inactive. When an action is enabled and triggered, it deactivates its respective wait mode 6106 and an avail mode 6108.

DCGs expose hidden interactions between coordinators. As shown in Fig. 61B, DCG 6102 reveals that an interaction between $wait_b$ and $wait_c$, of wait modes 6106, occurs outside of the rendezvous coordinator with two-participant preemption. The interaction is shown as a combination of edges 6110 and 6112, and preempt

action node 6114. As shown, the interaction might cause *wait*_c to intercept *wait*_b's wait for the resource.

A. *Partitioning SCGs and DCGs Across Action-Only Barriers*

5   A useful static graph transformation that can be performed on both SCGs and DCGs is partitioning across action-only barriers. An action-only barrier is a barrier within the system across which state cannot be maintained. Fig. 62A shows a communication channel between partitions of an SCG 6200 that can cause an action-only barrier. Fig. 62B shows a DCG 6202, corresponding to SCG 6200 after

10   partitioning across the access-only barrier. In applying this to DCGs, all transient edges are left unchanged.

     To perform the transformation across the action-only barrier the following steps are performed. First, the nodes within SCG 6200 that will be placed on each side of the action only barrier (in other words, perform a graph cut across the barrier,

15   as in Fig. 62A) are identified. Second, each constraint edge that crosses the action-only barrier is replaced with an appropriate template. Fig. 63 depicts constraint edges 6300 and 6302 that cross the action-only barrier, from Fig. 62A, and their corresponding templates 6304 and 6306. Finally, an action node created in the last step is filled in with an action sensitive to the activation (or deactivation) of the

20   disjunctive node d opposite the incident edge.

B. *Action/Constraint Conflicts*

     It is possible for an action to conflict with stable states during the execution of a software system. If the action has higher priority than a constraint on the stable

25   state, then a glitch will occur whenever the situation is exercised. The glitch infers a system state that may, or may not, actually be entered. However, the effects of the glitch can be propagated through the rest of the software system, even though the system state inferred by the glitch has not been entered. On the other hand, if the relevant constraint has a higher priority than the action, then the action cannot be

30   performed in a given system configuration and may be a candidate for removal from the software system.

100

C.   *Action/Action Conflicts*

An action can sometimes conflict with another action. Action/action conflicts are detected using similar checks to those given for stable-state consistency. These checks may occasionally leave two or more triples with overlapping modes and

5   triggers but with contradictory actions. These contradictory actions can be resolved in a number of ways, some static and some dynamic. A static solution to the contradictory actions is to conservatively eliminate any possible conflicting portions from one of the triples. An example of a runtime solution is to allow the conflicting actions to propagate throughout the rest of the triples, resolving the conflict based on

10   priority when it is time to lock down the new configuration.

4.   *Model Checking*

SCGs and DCGs can be used directly for a wide variety of checks involving what is allowed in particular configurations and what effects can result from changes on a small scale. However, for checking properties that may span several different

15   system configurations in a sequence, other transformations can perform these checks more efficiently.

Model checking typically describes techniques in which finite state system models are checked against predicates in temporal logic. Control graph configurations, both SCGs and DCGs, represent systemwide state for a software

20   system; it therefore follows that many abstract control graph properties can be verified using standard model checking. A binary decision diagram (BDD) can be a compact, though not always optimal, representation of transition relations of an extremely large state machine. BDDs frequently provide representations of systemwide state that are logarithmic relative to the size of the software system's

25   statespace. SCGs and DCGs are also compact representations of large statespaces. However, there are many standard checks that can be easily performed on BDD representations that would be difficult to perform on a SCG or DCG representation of a software system.

To take advantage of the standard checks available for BDDs, a preferred embodiment of the current invention includes a system and method for converting SCGs and DCGs into BDDs, without incurring the penalty of fully elaborating the state space of the software system.

5 ### A. _Temporal Unrolling_

Fig. 64 depicts a current DCG 6400 along with a next DCG 6402, which is the result of temporally unrolling DCG 6400. With reference to Fig. 64, a temporal line 6404 separates current DCG 6400 from next DCG 6402. Temporal unrolling of

10 current DCG 6400 allows a current control graph configuration, embodied in current DCG 6400, to be related to a set of next configurations, embodied in next DCG 6402, of the DCG for a particular software system. Unrolling current DCG 6400 across temporal line 6404 involves making a copy, to the right of temporal line 6404, of all disjunctive nodes, constraints, and all nondelayed actions within current DCG 6406.

15 For clarity, a prime is added to the label of each copied element. For each delayed action within current DCG 6400, a sensing edge 6406 is connected to each appropriate nonprimed node 6408, and an enforcing edge 6410 is connected to each appropriate primed node 6412. When a sensing edge is an event edge, some additional nodes must be created. For an event sensing edge 6414, a new disjunctive

20 node 6416, which represents the event itself, is created. A new conjunctive node 6418 tying new conjunctive node 6418 back to a creating node 6420 that created the event, constraining new conjunctive node 6418 state and its primed state appropriately (_e.g._, a+f event can only occur when f is already deactivated, but it leaves f' activated, as shown).

25 This unrolled DCG has the characteristic function:

$$f_c = (\neg a \vee \neg b \vee c) \wedge (\neg a' \vee \neg b' \vee c') \wedge (\neg f \vee \neg +f) \wedge (\neg +f \vee f') \wedge (\neg e$$
$$\vee \neg +f \vee b') \wedge (\neg e \vee \neg +f \vee \neg g').$$

102

This function now encodes the relation between current configuration and next configuration, which is annotated

$$C \xrightarrow{\ c\ } C'$$

5

for configurations C and C'.

Fig. 65A depicts a simple DCG 6500. Fig. 65B depicts an unrolled DCG 6502 for simple DCG 6500. With reference to Figs. 65A and 65B, the characteristic function of unrolled DCG 6502 is as follows:

10 $fc = (\neg a \lor c) \land (\neg c \lor \neg + c) \land (\neg + c \lor \neg a') \land (\neg a' \lor c') \land (\neg + c \lor c'),$

and its transition relation is enumerated in Table 7.

Table 7: A transition relation for the characteristic function of unrolled DCG 6502.

C			C'	
a	c	$+c$	a'	c'
0	0	0	0	0
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	1	0	0	0
0	1	0	0	1
0	1	0	1	1
1	1	0	0	0
1	1	0	0	1
1	1	0	1	1

B. _Computation Tree Logic_

5   Application of model checking requires a predicate logic that is powerful enough to express the temporal state relationships that will be checked. Computation tree logic (CTL) is a superset of the temporal logic introduced above, and it includes two new operators: X (for neXt) and U (for Until). A X Φ is true in the present if Φ is true in all possible next states. E X Φ is true in the present if Φ is true in at least

10   one of the possible next states. A (Φ_0 U Φ_1) is true in the present if, on all outgoing paths, Φ_0 is true until Φ_1 becomes true.

        The lambda calculus provides a flexible form for representing temporal logic expressions. Lambda expressions (_i.e.,_ expressions taking the form $\lambda x.E$) are often used to represent functionals, or functions that operate on other functions. The

15   expression $\lambda x.E$ p represents the expression E with p replacing each occurrence of x. For example, $\lambda x.(x + y)$ 3 is equivalent to $(3 + y)$, and $\lambda x.(x + y)$ $3y + z$ is equivalent to $(3y + z + y)$ or $(4y + z)$.

        To make use of this with CTL, first consider that

20     $S \models E X \Phi$

is equivalent to

$\exists y(S \xrightarrow{c} y \land y \models \Phi)$;

25

the notation

$S_a \models \Phi$

indicates that Φ is true in state S_a.

5   Therefore, a lambda expression representing all configurations for which predicate Φ is true on the next step on some path would be:

$$E \, X \, \Phi = \lambda x. \exists C.((x \xrightarrow{\;c\;} C) \wedge (C \models \Phi))$$

10 This defines $E \, X \, \Phi$ as a functional that can be applied to a configuration Z and that evaluates to true if and only if

$Z \models E \, X \, \Phi$.

15 Using this, along with boolean functions representing the transition relation and all configurations for which Φ is true, we can derive an additional boolean function that represents all configurations for which $E \, X \, \Phi$ is true.

Consider again the graph in Fig. 65. The characteristic function of unrolled DCG 6502 is

20

$$f_c = (\neg a \vee c) \wedge (\neg a' \vee c') \wedge (\neg c \vee \neg a').$$

To find a function that represents all configurations such that

E X Φ where Φ equals *a* ∧ *c*, we have:

$$EX(a \wedge c) \;=\; \exists C'.(fc \wedge C' \models (a \wedge c))$$

$$=\; \exists C'.(fc \wedge (a' \wedge c'))$$

$$=\; \exists C'.((\neg a \vee c) \wedge (\neg a' \vee c') \wedge (\neg c \vee \neg a') \wedge a' \wedge c')$$

$$=\; \exists C'.(\neg a \wedge \neg c \wedge a' \wedge c')$$

$$=\; \neg a \wedge \neg c$$

And so, using the lambda calculus version of

E X (*a* ∧ *b*)

and some boolean manipulation, we have obtained an expression

(¬a ∧ ¬b)

that is equivalent to the expression

E X (*a* ∧ *b*)

in terms of this particular control graph.

A fixed point of a Lambda expression (L) is an expression (F) such that

L F = F. For example, if

$$L = \lambda x.(x \vee y) \text{ and } F = \text{true}, \; L \; F = \text{true} \vee y = \text{true},$$

and so true is a fixed point of L. The least fixed point of Lambda expression $\lambda x.E_x$ is notated as $\mu x.E_x$, and the greatest fixed point is notated $vx.E_x$. Using these, we can describe two more functional representations:

$$E \; F \; \Phi = \mu Y.(\Phi \vee E \; X \; Y)$$

$$E \; G \; \Phi = vY.(\Phi \wedge E \; X \; Y)$$

C. *Binary Decision Diagrams (BDDs)*

The weakest link of symbolic model checking is the fact that boolean manipulation is *NP-hard* and can require large amounts of space in a computer's memory. However, it has been shown that for a large class of problems, binary decision diagrams can provide efficient representations for state machine transition relations that are easy to manipulate and combine. A BDD is a reduced representation of a truth table for a boolean function. Although any boolean function can be represented as a truth table, the truth table is exponential in size relative to the number of variables in the boolean function. A truth table can be represented as a tree in which each boolean variable corresponds to nodes and each assignment of values corresponds to a directed edge. Fig. 66A shows a truth table 6600 for a boolean "and" function 6602. Fig. 66B shows a truth tree 6604 that corresponds to truth table 6600.

Finding the truth value of an assignment is performed as follows. Starting from the root, edges are traversed until a leaf node is reached. The particular edge traversed from each node is labeled with the value assigned to the corresponding boolean variable. The leaf node reached contains the truth value of the assignment.

Figs. 67A, B, and C show several reduced BDDs 6700, 6702, and 6704 for truth tree 6604, shown in Fig. 66B. The procedure for looking up truth values is to

start at the root and traverse the edges labeled with the value assigned to the corresponding variable.

Whereas the number of nodes in a truth tree grows exponentially with the number of variables in a corresponding function, a BDD that grows only polynomially

5 can often be found. Reduction of a BDD is performed with the aid of a cache, where reduced subgraphs can be stored. This cache can be a hash table. In the reduce algorithm, as disclosed in BRYANT, R. E., "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys* 24, 3 (September 1992), 293-318, assume that each BDD node is a tuple (n, 1, r), where n is the name

10 of the corresponding variable, 1 is the BDD connected by the left edge, and r is the BDD connected by the right edge. Furthermore, a BDD cache has two operations — (1) put(BDD), for placing BDDs in the cache, and (2) lookup(name, BDD, BDD) (where the BDD parameters are the 1 and r subgraphs of the cached BDD) — for finding BDDs already cached and returning null if none is found.

15 Thus a BDD can be created for any boolean function by enumerating its truth table in the form of a truth tree and calling the "reduce" algorithm. However, this approach still suffers from exponential growth, since it requires exponential space for the truth tree until the BDD is constructed. However, BDDs can be efficiently and methodically grown using a procedure embodied in the "apply" algorithm as disclosed

20 in BRYANT, *Id.* Apply generates a BDD that represents an arbitrary boolean operation applied to two BDDs, *i.e.,* B_r such that $B_r = B_1 op B_2$.

The apply algorithm is derived from Shannon expansions of the functions represented by the input BDDs. A Shannon expansion represents a boolean function as an expression containing partial evaluations. For example, expanding f(x, y, z) in

25 terms of x:

$$f(x, y, z) = (x \wedge f(1, y, z)) \vee (\neg x \wedge f(0, y, z))$$

108

which can, of course, be further expanded in terms of y and z. It is typically notated

$$f = (\neg x \wedge f|_{x \leftarrow 0}) \vee (x \wedge f|_{x \leftarrow 1}).$$

5   This is also known as the cofactor expansion of a function. A Shannon expansion of f op g is:

$$f \text{ op } g = \neg[x \wedge (f|x \leftarrow 0 \text{ op } g|x \leftarrow 0)] \vee [x \wedge (f|x \leftarrow 1 \text{ op } g|x \leftarrow 1)]$$

10

Algorithm 3 BDD *reduce*

BDD reduce (*T*)
Require: *T* is a BDD
    $C \leftarrow$ **new** BDD cache
15   C.put(new BDD(1, **null**, **null**)) // *cache leaf nodes*
    C.put(new BDD(0, **null**, **null**))
    return reduceC (*T, C*)
end

20 **BDD** reduceC(*T, C*)
Require: *T* is a BDD, and C is a BDD cache
        if T = **null** then
                return null
        else
25            $l \leftarrow$ reduceC(*T.l, C*)
            $r \leftarrow$ reduceC(*T.r, C*)
        end if
        return *C:lookup(T.n, l, r)*
end

30

Algorithm 4 BDD *apply*

BDD apply (*op, B₁, B₂*)
35 **Require:** *op* is an operator; *B₁* and *B₂* are BDDs
        $C \leftarrow$ **new** BDD cache
        C.put(1, **null**, **null**)) // *cache leaf nodes*
        C.put(0 **null**, **null**)
        return applyC(*op, B₁, B₂, C*)
40 **end**
BDD applyC(*op, B₁, B₂, C*)
        if depth *(B₁, n)* = depth(*B₂.n*) **then** // *Same variable*

$n \leftarrow B_1, .n$

$l \leftarrow$ applyC $(op,\ B_1,\ l,\ B_2,\ l,\ C)$

$r \leftarrow$ applyC $(op,\ B_1,\ r,\ B_2,\ r,\ C)$

else if depth $(B_1.n) <$ depth$(B_2.n)$ **then** // $B_1\ precedes\ B_2$

5      $n \leftarrow B_1, .n$

$l \leftarrow$ applyC $(op,\ B_1,\ l,\ B_2,\ C)$

$r \leftarrow$ applyC $(op,\ B_1,\ r,\ B_2,\ C)$

else // $B_2\ precedes\ B_1$

$n \leftarrow B_2.n$

10      $l \leftarrow$ applyC $(op,\ B_1,\ B_2,\ l,\ C)$

$r \leftarrow$ applyC $(op,\ B_1,\ B_2,\ r,\ C)$

end if

return $C.lookup(n,\ l,\ r)$

end

15

D.   *BDD Representations of Control Graphs*

A BDD can often be used to represent transition relations with exponential
20   state-spaces while using only polynomial storage space. A BDD can be constructed
from a control graph by unrolling the control graph, as described above, and then
using the apply algorithm, described above, to build a BDD from a characteristic
function for the unrolled control graph. To efficiently represent the state-space,
attention must be paid to variable ordering within the characteristic function. Good
25   orderings have contiguous sequences of highly correlated variables.

Consider DCG 6102, which represents the unrolled three-client rendezvous/
preempt coordinator, shown in Fig. 61. This example is interesting because the state-
space represented grows exponentially with the number of participants. BDD
representations would be of little value for our purposes if they experienced
30   exponential growth in creation or if they required exponential storage. As shown
below, BDDs provide compact representations for such examples.

Fig. 68 shows the results of using the apply algorithm to grow a BDD 6800,
which represents the characteristic function of unrolled DCG 6502 from Fig. 65.

An unrolled DCG contains a great deal of information that can aid a designer in finding an efficient variable ordering. Constraints and pseudo-constraints *(i.e.,* those introduced by unrolling) connect variables that are likely to have the least change with respect to each other. For example, given the following constraints:

5

$$\{..., C \Rightarrow \neg G, C \Rightarrow B, B \Rightarrow C, ...\}$$

if $C \Rightarrow \neg G$ is uncontested, we know that C and G are fairly strong candidates to be located next to each other in the variable ordering. However, $C \Rightarrow B$ and $B \Rightarrow C$

10   suggest that B and C are even stronger candidates.

Fig. 69A shows an unrolled rendezvous DCG 6900. With reference to Fig. 69A, unrolled rendezvous DCG 6900 is created by temporally unrolling DCG 6102, which represents the rendezvous coordinator, as described with reference to Fig. 61. The characteristic functions for unrolled rendezvous DCG 6900 are as follows:

15

$$f_a = \text{wait}_a \wedge (\neg\text{wait}_b \wedge \neg\text{wait}_c) \wedge \text{avail} \Rightarrow \neg\text{wait'}_a, \neg\text{avail'}$$

$$f_b = \text{wait}_b \wedge \neg\text{wait}_c \wedge \text{avail} \Rightarrow \neg\text{wait'}_b, \neg\text{avail'}$$

$$f_c = \text{wait}_c \wedge \text{avail} \Rightarrow \neg\text{wait'}_c, \neg\text{avail'}$$

20   The DCG shows that wait_xs could be strong candidates for colocation with their respective wait'_xs, with wait_a, wait'_a, wait_b, wait'_b, wait_c, wait'_c, avail, avail' as a reasonable order. The above variable ordering yields a BDD that has fifteen nodes, which can be referred to as a temporal cluster.

For further improvement, notice that each of the primed wait nodes depends
25   upon unprimed versions of wait nodes adjacent to and below it in the graph *(e.g.,* wait'_a depends upon wait_a, $\neg\text{wait}_b$, and $\neg\text{wait}_c$). This suggests that wait_a, wait_b, and

wait_c should all precede wait'_a in a BDD, wait_b and wait_c should precede wait'_b, and so forth. This, combined with the order constraint suggested in the last paragraph, indicates that wait_c, wait'_c, wait_b, wait'_b, wait_a, wait'_a, avail, avail' should be a very good ordering. In fact, it yields a BDD with eleven nodes. This variable ordering will be referred to as cluster/depend. While cluster/depend is only slightly better than the order given in the previous example, it offers an advantage when the number of rendezvous wait participants increases from three. As shown in Table 8, the simple temporal cluster order creates BDDs that consistently have around four times the number of nodes as there are variables. The cluster/depend order creates BDDs that are only around twice the number of participants. Both, however, are linear in size based on the number of participants. This is much better than the boolean expression form of the characteristic function, which grows quadratically with the number of participants.

Table 8: Growth rate for BDDs representing temporally unfolded rendezvous.

wait participants	temporal cluster	cluster/depend
3	15	11
8	35	21
15	63	35
25	103	55
50	203	105

Fig. 69B shows that the critical factor in ordering for this DCG is really just the order of wait_c, wait_b, and wait_a with respect to each other. The ordering shown in Fig 69B is equivalent in quality to cluster/depend in quality.

With these compact and canonical symbolic representations of rendezvous and other coordinators' exponential statespaces, we can apply model checking techniques to perform preemptive debugging and catch many bugs before implementation software is synthesized, compiled, and run.

112

E. *Application of Model Checking*

Model checking is performed using McMillan's *AndExists* algorithm, disclosed in, McMILLAN, K.L., *Symbolic Model Checking: An Approach to the State*

5   *Explosion Proble*m, Ph.D. thesis, Carnegie Mellon University, 1992. AndExists evaluates

$$\lambda x. \exists V.(p \wedge q)$$

10   where p and q are boolean expressions represented as BDDs and V is a boolean assignment vector

$$(v_i \in \{\mathbf{false}, \mathbf{true}\}).$$

15   In conjunction with

$$E\ X\ \Phi = \lambda x. \exists C.((x \xrightarrow{c} C) \wedge (C \models \Phi))$$

the above allows $E\ X\ \Phi$ to be computed.

20   Many system properties can be verified by checking multiple components simultaneously. Some examples for deadlock

$$(\exists S : S \models A\ G\ \mathbf{false})$$

25   and livelock

$(\exists S : S \mid = A \ F \ S)$.

An important check is determining whether a software system always
converges on a consistent state. When a DCG is partitioned among several

5   subsystems *(e.g.,* on a multiprocessor architecture), an action-only barrier is formed
between the portions on each subsystem. Frequently, the actions that cross have
delays that can span several scheduling steps, and these delays are often functions of
bus traffic or other factors. To make such a system match the semantics described
here, it is necessary to synchronize all subsystems so that action delays are never

10   more than a single scheduling step. However, this is too conservative for an
execution model, and it eliminates some of the advantage of multiprocessor
architectures. It often makes more sense to let components interact asynchronously
when possible and to ensure that the system will always resolve to a consistent state.
To do so, it is necessary to choose an execution model that represents asynchrony.

15   Interleaving asynchronous models assume that processes can change state at
any time—but only one at a time. However, this assumption may be too conservative,
since several correlations can be made between state-bits on a single component. It is
more accurate to say that any component can perform any locally legal control state
change at any time. As shown in Figs. 70A, B, C, and D, this means that one

20   moment after the configuration shown in Fig. 70A is valid, either of the
configurations shown in Fig. 70B or Fig. 70C can be valid. However, the
configuration shown in Fig. 70D cannot be valid, because it reflects two simultaneous
state changes. However, since these changes are concurrent, the configuration in Fig.
70D may result after two time steps of the configuration in Fig. 70A.

25   *F.*   *Look-Ahead Predicates*

Using model checking to perform inquiries into a particular property in a DCG
produces a boolean function that identifies all configurations for which it is true.

In debugging a system, designers want to track whether configurations in a

30   particular execution can possibly lead to a configuration in which a particular

predicate holds and which configurations would do so. The expressions derived from the application of model checking are look-ahead predicates.

Control/Dataflow Graphs (CDGs)

5   Fig. 71 shows a control/dataflow graph (CDG) 7100. CDG 7100 represents dataflow-based transparent actions 7102 and 7104 as part of its structure. CDG 7100 has the same overall structure as a DCG but further allows both data ports 7106 and 7108 and dataflow nodes 7110 and 7112. Dataflow actions are allowed, within this structure, to cause control changes.

10

A CDG is a triple, $G = (\Gamma f, \Delta f ; E f)$, in which:

- Γf is a set of conjunctive and action nodes, $i.e.$, $\Gamma f \subseteq (C \cup A)$, in which C is the universe of all possible conjunctive nodes and A is the universe of

15   all possible actions.

- Δf is a set of disjunctive and dataflow nodes, $i.e.$, $\Delta f \subseteq (D \cup F)$, in which D is the universe of all possible disjunctive nodes and F is the universe of all possible dataflow nodes.

20

- $Ef \subseteq [\{Tf, Tt\} \times \{Hf, Ht\} \times \{T, N\} \times ((\Gamma f \times \Delta f) \cup (\Delta f \times \Gamma f))]$ is a set of directed edges that are labeled either Transient (T) or continuous (N).

A.   *Transaction/Constraint Conflicts*

25   Transactions are part of the definition of coordinators. However, they are not usually executed directly by the coordinator. Transactions may be initiated by

components, and responses must often be generated by components rather than by the coordinator itself. Coordinators contain constraints that can play a role in enforcing these semantics, and it is important to ensure that a component's constraints are consistent with the transactions specified for the coordinator.

5   1.   Dedicated RPC

Fig. 72 shows a CDG 7200 representation of an RPC system. CDG 7200 does not consider any coupling between control and data. There are two aspects to control. The first aspect of control is steady-state control, in which there is global control of transitions from one state to another. The second aspect of control is the

10   interaction between control and data. Control typically must consider how the software system deals with the transfer of data, in the form of parameters, for the recipient and return values.

B.   *Partitioning CDGs Across Message-Only Barriers*

15   Just as SCGs and DCGs can be partitioned across action-only barriers through template-based graph transformations, both they and CDGs can be partitioned across barriers that permit only message traffic. The first stage in this transformation is as described above with respect to SCGs and CDGs. Fig. 73A shows the second step in partitioning a CDG 7300 across an action-only barrier 7302. With reference to

20   Fig. 73A, the second step is to transform each action on the barrier into supplementary actions and messages. Fig. 73B shows the graph of DCG 6202, from Fig. 62B, with the action-only barrier transformed to a message-only barrier 7304.

C.   *Dataflow Consistency and SDF Extraction*

25   Dataflow consistency means that when the system is viewed as a whole, production rates are compatible with their related consumption rates. A system that is consistent in terms of dataflow will have no configuration in which production exceeds consumption on any path. Failure to detect such inconsistencies in advance could result in difficult to track memory leak bugs and bugs wherein data is dropped

30   or overwritten before it is processed (depending on queue-management policies).

Synchronous dataflow (SDF) extraction for these dataflow graphs (DAG) is performed in two parts. First, a constant rate sub-DAG is fenced off through a pair of cuts. A constant-rate sub-DAG is a directed acyclic dataflow graph in which each port has constant token rates. Given a constant rate sub-DAG, each edge is a linear

5   relation between two dataflow actions. Second, by solving a consistent series of linear equations, schedule components can be found for each node. Fig. 74A shows a CDG 7400 with a set of message rate guarantees 7402. Fig. 74B shows a dataflow graph 7404 based on CDG 7400. With reference to Figs. 74A and B, the start of each edge 7406 is labeled with a production rate 7408 and the sink is labeled with a

10   consumption rate 7410. While inconsistent series of equations yield no solution, they also represent inconsistent dataflow. Hence, one algorithm can be used for both dataflow consistency and SDF extraction.

1.   Finding Normalized Schedule Coefficients

Schedule coefficients indicate the number of times a particular component is to

15   be executed sequentially to consume all tokens generated by a component earlier in the dataflow graph. The problem is now to find a set of minimum practical scheduling coefficients. For dataflow actions a, b, c, and d (shown in Fig. 74B) the minimum scheduling coefficients are notated A_0, B_0, C_0, and D_0. Practical schedule coefficients must be natural numbers because it is typically not possible to execute a

20   component a fractional number of times. However, in obtaining these we need an intermediate step using rational relative schedule coefficients. Relative schedule coefficients are the minimum coefficients divided by a particular component's coefficient. The schedule coefficient for component a normalized to component d's coefficient is notated A_d. The value of D_d is defined to be 1.

25   Finding relative schedule coefficients is accomplished by deriving a set of relative scheduling constraints between pairs of nodes. Each internal edge in a constant rate cluster must have constant token rates on each side. The relative rates for two nodes, x and y, connected by an edge with token rates of m on the x side and n on the y side is $nx = my$, which means that whenever x fires n times, y must fire m

30   times. We can then relate their scheduling coefficients as $mX = nY$. This

relationship holds for all valid practical and relative scheduling coefficients, not just the minimum coefficients.

Normalized scheduling rates can be expressed in terms of a single component reference. To do so, each set of relative rates is first expressed as a linear equation. For example, let A_z be the normalized schedule coefficient for component a and B_z be the normalized schedule coefficient for component b. We express their relative coefficient relationship as $nA_z - mB_z = 0$. And so, for the relative ratio matrix -R, we are solving for the schedule coefficient vector in terms of z or

$\hat{\Gamma}_z$, with $-R \cdot \hat{\Gamma}_z = \hat{1}_z$, where $\hat{1}_z$

is a vector that is zero in all except the z position.

For dataflow graph 7404, using d as the reference component this results in the following:

$$R = \begin{pmatrix} 3 & 0 & -2 & 0 \\ 0 & 4 & -3 & 0 \\ 0 & 0 & 5 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$1_d = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Solving $R \bullet \Gamma_d = 1_d$ for Γ_d, we have:

$$\Gamma_d = \begin{pmatrix} A_d \\ B_d \\ C_d \\ D_d \end{pmatrix} = \begin{pmatrix} 4/15 \\ 3/10 \\ 2/5 \\ 1 \end{pmatrix}$$

5

Notice that the top three rows of $-R$ correspond to the three edges in dataflow graph 7404.

Notice that $\hat{\Gamma}_d$ has a solution if and only if the dataflow graph is consistent. Graph inconsistency is only possible if there are at least as many edges as actions, 10 *i.e.,* if the system of equations is overdefined. If Gaussian elimination is used for solving the matrix system, then the goal with an overdefined system is to cancel out one row (typically the bottom), making that row all zeros. If the dataflow graph is inconsistent, then the result is dataflow graph 7404 will have a row of the form [0 0 0 0...0|x], where $x \neq 0$.

15   Fig. 75 shows a dataflow graph 7500. With reference to dataflow graph 7500, the starting Gaussian matrix is as follows:

$$\begin{pmatrix} 5 & -2 & 0 & 0 & \bigm| & 3 \\ 1 & 0 & -2 & 0 & \bigm| & 15/2 \\ 0 & 2 & 0 & -4 & \bigm| & 3/2 \\ 0 & 0 & 2 & -3 & \bigm| & 15/4 \\ 0 & 0 & 0 & 1 & \bigm| & -11/4 \end{pmatrix}$$

After elimination and attempting to zero out the bottom row, the result is as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \bigm| & 3 \\ 0 & 1 & 0 & 0 & \bigm| & 15/2 \\ 0 & 0 & 1 & 0 & \bigm| & 3/2 \\ 0 & 0 & 0 & 1 & \bigm| & 15/4 \\ 0 & 0 & 0 & 0 & \bigm| & -11/4 \end{pmatrix}$$

5

The result shown above indicates that the system represented by dataflow graph 7500 cannot be properly scheduled. Furthermore, even without exact production and consumption rates, a similar procedure can be use to catch inconsistent dataflow.

2.   Practical Schedule Coefficients

10   Executing a system requires all scheduling coefficients to be integers. This can be facilitated by finding a lowest common denominator (LCD) of a set of normalized scheduling coefficients and multiplying the LCD through. For Fig. 74B, the LCD is 30, and so:

15

$$\Gamma_0 = \begin{pmatrix} A_0 \\ B_0 \\ C_0 \\ D_0 \end{pmatrix} = \begin{pmatrix} 8 \\ 9 \\ 12 \\ 30 \end{pmatrix}$$

3.   Schedule Ordering

The order of components in the schedule must be consistent with causality in the software system being modeled. A proper ordering can be determined by a

20   topological sort of the dataflow graph *(e.g.,* b, a, c, d for the example in Fig. 74B). By combining scheduling coefficients, represented by vector Γ_0, with ordering, a

complete consistent schedule can be found. For dataflow graph 7404, the complete consistent schedule is as follows: 9b · 8a · 12c · 30d. This means execute action b 9 times while buffering results, then execute a 8 times while buffering results, then execute c 12 times while buffering results, and finally execute d 30 times.

5   It will be obvious to those having skill in the art that many changes may be made to the details of the above-described embodiment of this invention without departing from the underlying principles thereof. The scope of the present invention should, therefore, be determined only by the following claims.